KF-Ray,
Raytracer paralléle

Rapport de projet long ELI 4

Karin AIT-SI-AMER & Florian DANG

Polytech'Paris-UPMC

19 mai 2009

Remerciements

Nous tenons particulièrement à remercier notre tuteur Jean-Baptiste Mouret qui nous a supervisé tout au long du projet ainsi que Pierre Fortin dont les cours et les travaux pratiques nous ont permis de mener à bien la parallélisation de notre programme.

Nous remercions également Ludovic Saint-Bauzel qui s'est révélé être une aide précieuse pour les conseils précieux qu'il nous a donné au cours du projet ainsi que Cécile Braustein qui nous a initié aux mystères des logiciels de déboggage.

Introduction

Salle obscure et comble, confortablement installé dans votre siège, une boîte de pop-corns à la main, vous êtes plongés dans le film projeté sur une toile géante où Rémy et Linguini trouvent le moyen de se lancer des répliques tout en effectuant des cascades qui provoquent le rire chez tous les spectateurs. Un film banal donc. A une exception près. Nos deux comparses Rémy et Linguini ne sont pas des acteurs ordinaires. Ils ne sont autre que les héros d'un tout nouveau film d'animation pondu comme il se doit par le studio Pixar : Ratatouille. Ce qui signifie qu'ils ne sont en réalité que des acteurs purement virtuels, tout droit sortis des moteurs de rendu du studio d'animation. Après Toy Story, 1001 pattes, Shrek et Final Fantasy, Ratatouille vient une nouvelle fois illustrer de manière éclatante les progrès accomplis en matière de rendu graphique.
Désormais, les rendus dits “photoréalistes” sont partout. Après avoir envahi le monde des effets spéciaux du cinéma, de la publicité et du design, ils s'invitent dans les films d'animation ou encore l'imagerie médicale.

La recherche dans ce domaine n'a jamais été aussi intensive qu'actuellement et d'aucun diront que l'imagerie de synthése, épaulée par des moteurs de rendu graphique de plus en plus puissants à un avenir tout tracé devant elle.

Cependant, les logiciels permettant de mettre en oeuvre les techniques de rendu d'image sont plus la très grande majorité réservés à des professionnels ce qui implique un coût à l'achat extrêmement élévé. Pourtant, il existe bien un logiciel libre complet - POV (Persistance Of Vision) - qui s'appuie sur le principe du raytracing. Malgré son développement très abouti, POV n'a pas été conçu pour permettre d'exécuter cet algorithme en parallèle sur un parc de machines (il existe bien des patchs mais ceux-ci ne s'avèrent pas concluants). Lorsque l'on considère la complexité du programme - il peut parfois falloir plusieurs heures pour obtenir le rendu d'une image particulièrement complexe - il s'agit d'un défaut qui peut avoir son importance.

Notre but n'était pas de concurrencer POV constitué d'une équipe de développeurs expérimentés, mais de mettre en place un logiciel de raytracing léger, d'une prise en main intuitive, capable de mener des calculs en parallèle sur plusieurs ordinateurs afin d'obtenir des images de même qualité que POV avec un moindre coût en temps (grâce au parallélisme).

Table des matières

1 Organisation du projet **8**
1.1 Les différents modules . . . 8
1.2 Les choix de programmation . . . 8
1.2.1 Le langage C . . . 8
1.2.2 GTK+ . . . 9
1.2.3 MPI . . . 9
1.3 Les outils . . . 9
1.3.1 Subversion (SVN) . . . 9
1.3.2 Lex et Yacc . . . 10
1.3.3 Code : :Blocks et organisation du code source . . . 10
1.3.4 Doxygen . . . 10
1.3.5 Site de développement sur Google Code . . . 10
1.3.6 Site de présentation sur http ://kfray.free.fr . . . 11
1.4 Répartition des tâches et échéancier . . . 12

2 Parser & Interface graphique **13**
2.1 Format inital du fichier de description de scène . . . 13
2.2 Les différents objets . . . 13
2.3 Format final du fichier de description de scène . . . 13
2.3.1 la scène . . . 14
2.3.2 Le fichier de description scène KFR . . . 14
2.4 Le parser . . . 15
2.4.1 définition d'une expression régulière . . . 16
2.4.2 Yacc . . . 17
2.5 Glade 2 et GTK+ 2 . . . 18
2.5.1 Description . . . 18
2.5.2 Réalisation . . . 20

3 L'algorithme de Ray-tracing **21**
3.1 Principe 21
3.1.1 Le modèle de l'appareil photo simplifié 21
3.1.2 L'algorithme 22
3.1.3 Quelques notions d'optique géométrique 23
3.1.3.1 l'optique géométrique 23
3.1.3.2 définitions et vocabulaire 24
3.1.3.3 les lois de l'optique géométrique 24
3.2 Ray Traycer 25
3.2.1 Premiers pas 25
3.2.2 Les rayons 26
3.2.2.1 Rayon initial 26
3.2.2.2 Rayon réfléchi 26
3.2.2.3 Rayon réfracté 26
3.2.2.4 Intersection rayon/objet 27
3.2.3 Modèles d'illumination 27
3.2.3.1 le modèle RVB 27
3.2.3.2 les sources de lumière 28
3.2.3.3 définition d'un modèle d'éclairement 28
3.2.3.4 modèles d'éclairement 28
3.2.4 Textures 33
3.2.4.1 texture procédurale 33
3.2.4.2 bump mapping 34
3.2.5 Anti-aliasing 35
3.2.6 La caméra 36
3.2.7 Recul sur le rendu 37

4 Parallélisme **39**
4.1 Première approche 39
4.1.1 Principe 39
4.1.2 L'art de découpage une image 40
4.2 Application sur KF-Ray 41
4.2.1 Equilibrage de charge dynamique 41
4.2.2 Résultats 44
4.3 Améliorations possibles de la parallélisation 44
4.3.1 Surcoût des communications 44
4.3.2 Faire travailler le maître 45
4.3.3 Auto-régulation (work-stealing) 45

A Exemple de fichier de scène complet **47**

B Intersection d'un rayon avec une sphère **48**

C Intersection d'un rayon avec un plan **50**

Table des figures

1.1 Dépôt SVN de Polytech 9
1.2 KF-Ray sur Google Code 11
1.3 Site de présentation : http ://kfray.free.fr 12
1.4 Planning de GANTT 12

2.1 Fichier de description de scène primitif 13
2.2 Graphe de collaboration de s_scene 14
2.3 format d'un fichier de description de scène 3D 15
2.4 The Regex Coach 17
2.5 Actions Yacc 18
2.6 Interface schématique 19
2.7 Frontend KF-Ray GUI 20

3.1 Modèle de l'appareil photo simplifié 22
3.2 Lois de Snell-Descartes 25
3.3 Sphère avec un raycast sans éclairement 26
3.4 Transparence 27
3.5 L'espace RVB 27
3.6 Modèle de la réflexion diffuse 28
3.7 Illustration de la réflexion 29
3.8 Sphère et modèle de Lambert 30
3.9 Modèle de Phong 30
3.10 Sphère et modèle de Phong 31
3.11 Effet de la rugosité pour une valeur petite de n 32
3.12 Modèle géométrique de Blinn-Phong 32
3.13 Sphère et modèle de Blinn-Phong 33
3.14 Illustrations des textures procédurales (turbulence, marbre, wood) 34
3.15 Effet du bump mapping 35

3.16 Observation du phénomène d'aliassage 36
3.17 Lancer de rayons parallèles puis vers un point de convergence 37
3.18 Scène similaire sur POV-Ray avec fichier de description de scène 38

4.1 Découpage d'un problème de parallélisation 40
4.2 Efficacité paralléle en découpage "naïf" 41
4.3 Graphique des performance de la parallélisation 44

Chapitre 1

Organisation du projet

1.1 Les différents modules

Dans un premier temps, il a été nécessaire de lister toutes les fonctionnalités souhaitées du programme afin de pouvoir nous partager le travail et rendre ces dernières les plus indépendantes les unes des autres. Notre programme doit donc comporter les modules suivants :

1. un loader afin de parser le fichier de description de scène d'entrée (nous avons opté pour les outils Lex et Yacc) ;
2. une fichier permettant de traiter le format de l'image obtenue (nous avons opté pour le format ppm) ;
3. un fichier permettant de manipuler les différents objets tels que les sources de lumière, les objets géométriques, les matériaux ainsi que les vecteurs ;
4. un fichier permettant de manipuler les textures ;
5. le raytracer en tant que tel ;
6. une interface GTK qui permet de manipuler de manière plus intuitive le programme.

1.2 Les choix de programmation

1.2.1 Le langage C

Un certain nombre de langages de programmation s'offrait à nous dont principalement le Java, le C++ et le C. Afin de pouvoir au mieux profiter du temps qui nous était imparti, il nous a semblé plus judicieux d'opter pour le C, langage que nous connaissons et que nous avons déjà utilisé pour mener des projets à bien. Cette solution nous a permis d'être plus au rapide au niveau du développement du programme notamment par l'utilisation quotidienne des logiciels de débogage tels que GNU Debugger (gdb), Valgrind ou encore Data Display Debugger (DDD). Par ailleurs, le C est réputé pour être plus rapide car plus proche du langage machine, argument qui nous concerne puisque la rapidité de notre raytracer est une de nos contraintes principale.

Notre code doit être relativement clair, cohérent afin que n'importe qui puisse reprendre, modifier et améliorer le programme facilement.

1.2.2 GTK+

Il était nécessaire de développer une interface graphique qui permettrait à l'utilisateur de charger simplement un fichier de description de scènes 3D et de pouvoir lancer, après avoir défini quelques options, le calcul du rendu de l'image. La librairie libre gratuite GTK+ libre est celle sur qui nous avons porté notre choix étant donné que nous avions une expérience avec. Par ailleurs, elle existe sur de nombreuses plate-formes comme Linux, Windows et MacOS. Comme elle est sous licence LGPL, nous serons libres de distribuer notre programme comme bon nous semble.

Cette interface se doit d'être intuitive, ergonomique et ludique. L'utilisateur qui le lance pour la première fois doit être capable de le prendre en main très rapidement.

1.2.3 MPI

L'algorithme du raytracer est très gourmand en temps de calcul. D'où la nécessité de pouvoir l'exécuter de façon parallèle sur plusieurs machines. Ces différents processus doivent se synchroniser et communiquer entre eux, ce qui est assuré en pratique par l'utilisation d'une bibliothèque qui permet la mise en oeuvre du modèle de programmation par échanges de messages : la bibliothèque MPI (Message Passing Interface).

1.3 Les outils

1.3.1 Subversion (SVN)

KF-Ray est un projet qui doit se mener sur une période de quatre mois qui demande une partie de développement conséquente ce qui implique une manipulation quotidiennt de nombreux fichiers et de lignes de code. La nécessité d'utiliser un outil nous permettant en tant qu'équipe de deux personnes, de pouvoir récupérer d'une manière simple et instantanée le travail effectué par l'autre s'est posée dès le début. Notre choix s'est porté sur Subversion (SVN), logiciel de gestion de sources et de contrôle de versions. Ce programme libre possède plusieurs fonctions dont notamment :

1. garder un historique des différentes versions des fichiers d'un projet ;
2. permettre le retour à une version antérieure quelconque ;
3. garder un historique des modifications avec leur nature, leur date, leur auteur...
4. permettre un accès souple à ces fichiers, en local ou via un réseau ;
5. permettre à des utilisateurs distincts et souvent distants de travailler ensemble sur les mêmes fichiers.

Notre dépôt SVN est l'emplacement central où sont stockées toutes les données relatives au projet KFRay. Un visualisateur du dépôt est accessible via cette URL distante : https ://svn.polytech.upmc.fr/viewcvs/aitsiame_tdang/

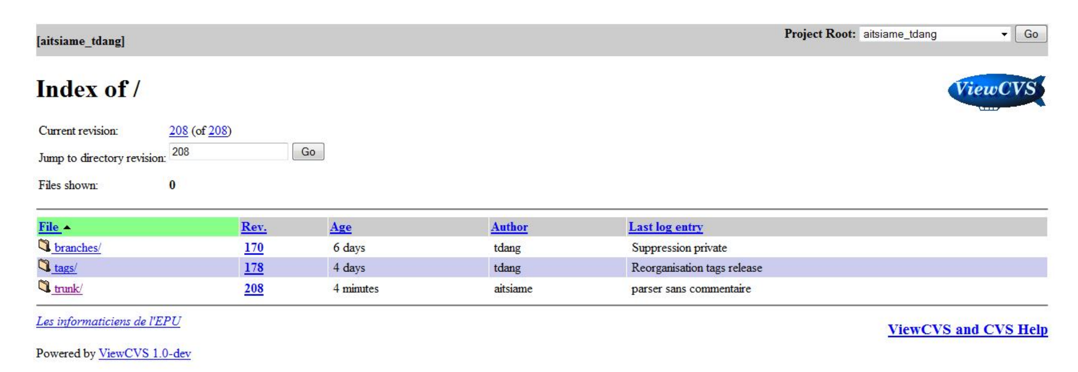


FIG. 1.1 – Dépôt SVN de Polytech

Un dépôt apparaît donc de l'extérieur comme un système de fichiers composé de répertoires au sein desquels on peut naviguer, lire et écrire selon les permissions accordées. Chacun des utilisateurs dispose d'une copie de travail : il s'agit d'un répertoir situé en local sur son poste et qui contient une copie d'une révision donnée.

Grâce à ce système, il nous a été aisé d'avancer tout les deux sur le projet de façon parallèle d'une manière très rapide. Ce système s'est révélé efficace et nécessaire tout au long du projet.

1.3.2 Lex et Yacc

Lex et Yacc sont des outils de génération d'analyseurs lexicaux (Lex) et syntaxiques (Yacc) en langage C, ce qui s'adapte particulièrement bien avec notre projet. L'analyse du fichier d'entrée se fait en deux étapes : la première est l'anlyse lexicale, qui fait partie du domaine de compétence de Lex, par l'intermédiaire de la fonction *yylex()*, qui se charge de consommer les terminaux (voir le chapitre consacré au parser). Elle va ensuite les signaler à l'analyseur syntaxique - Yacc - grâce à la fonction *yyparse()*.

1.3.3 Code : :Blocks et organisation du code source

Code : :Blocks est un environnement de développement gratuit et multi-plateforme (Windows et Linux). Conçu autour d'une architecture de plugins, il est extensible et configurable très facilement. L'un de ses principaux atouts est de pouvoir s'interfacer avec la plupart des compilateurs gratuits du marché. Par ailleurs, il possède quelques fonctionnalités qui accélèrent le travail du programmeur dans la mesure où il reconnaît les différents noms de variables et qu'il est aisé de retrouver une fonction d'un simple clic.

Nous avions quelques contraintes imposées concernant le code source. En effet, d'après une charte de code qui nous a été fournie en début de projet, aucune de nos fonctions ne doit dépasser vingt cinq lignes d'instruction et il ne doit pas y avoir plus de quatre vingt caractères par ligne. Nous avons, pour que l'indentation du code soit visible également sur Emacs ou VI, mis l'indentation en TAB et leur espace à 8 sous CodeBlocks.

1.3.4 Doxygen

Doxygen est un logiciel qui permet de générer une documentation du code source à partir de ses commentaires. Il est utlisé par de nombreuses équipes de développement comme celles de KDE, IBM et Mozilla. Son fonctionnement est d'une grande simplicité : la programmation s'effectue normalement mais le code est commenté d'une façon un peu particulière. Il ne reste plus qu'à configuer Doxygen - choisir le format de sortie (HTML, rtf, LaTeX ou XML), générer automatiquement des diagrammes... - pour obtenir une documentation exhaustive pour peu que l'on ait commenté toutes les fonctions du code.

1.3.5 Site de développement sur Google Code

L'ensemble du projet Google Code est le fruit du travail d'un certain Greg Stain qui s'interrogeait sur la meilleure manière de fournir aux projets libres une infrastructure adéquate. Pour inscrire un projet sous Google Code, il est nécessaire que le projet soit sous licence, nous avons opté pour la licence GNU GPL (GNU General Public License).



Fig. 1.2 – KF-Ray sur Google Code

Cette license fixe les conditions légales de distribution des logiciels libres du projet GNU. La principale caractéristique de la GPL est le *copyleft*, littéralement “copie laissée” qui consiste à “détourner” le principe du copyright pour préserver la liberté d’utiliser, d’étudier, de modifier et de diffuser le logiciel et ses versions dérivées. L’objectif de la licence GNU GPL - selon ses créateurs - est de garantir à l’utilisateur les droits suivants :

1. la liberté d’exécuter le logiciel pour n’importe quel usage ;
2. la liberté d’étudier le fonctionnement d’un programme et de l’adapter à ses besoins, ce qui passe nécessairement par l’accès au code source (d’où le présence de l’onglet *Download* au sein de notre page Google Code) ;
3. la liberté de redistribuer des copies ;
4. la liberté d’améliorer le programme et de rendre publiques les modifications afin que l’ensemble de la communauté en bénéficie.

Google code permet en outre - par le biais de l’onglet *issues* - à n’importe quel membre de la communauté de reporter des bugs ainsi que des améliorations éventuelles.

1.3.6 Site de présentation sur http ://kfray.free.fr

Nous avons décidé de concevoir notre propre site web comme vitrine de présentation de notre projet sur internet. Ce site présente le contexte dans lequel il a été mené, propose une gallerie d’image générées par notre programme, le manuel utilisateur, les délivrables, la documentation générée par Doxygen ainsi que nos sources.

En effet, nous souhaitons permettre aux personnes intéressées d’utiliser facilement notre programme qui comprend un manuel d’utilisateur.



Fig. 1.3 – Site de présentation : http ://kfray.free.fr

1.4 Répartition des tâches et échéancier

Il nous a été nécessaire de définir les différentes tâches à accomplir puis de nous répartir le “travail” de façon équitable. Nous avons alors conçu le planning suivant au tout début du projet :

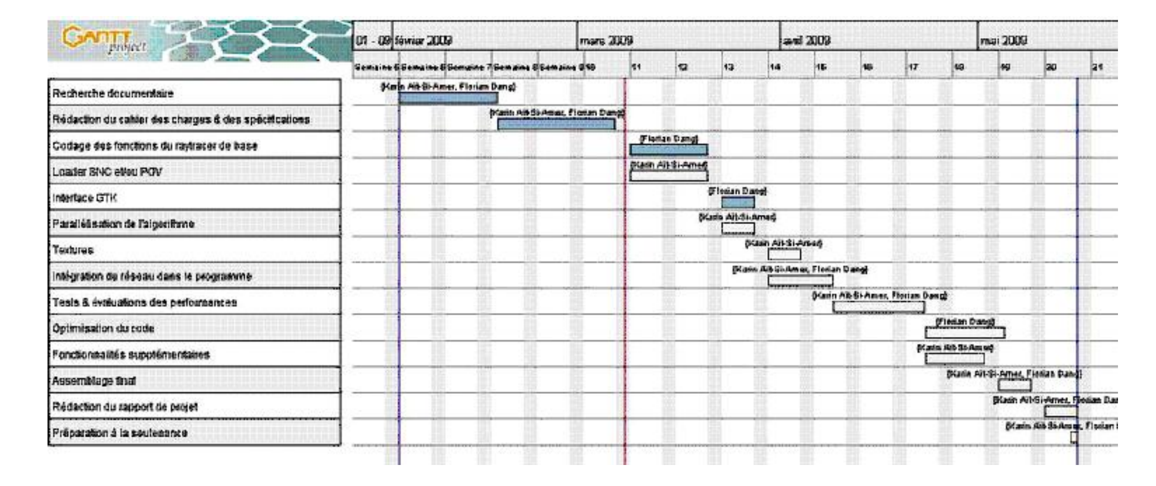


Fig. 1.4 – Planning de GANTT

Nous avons tenté de respecté ce planning. Face à la difficulté des problèmes techniques rencontrés (incompatibilite GTK/MPI qui nous a retardé par exemple), le planning a du être continuellement revu et modifié en n’oubliant pas de définir des priorités. Dans l’ensemble finalement, le planning reflète d’une manière satisfaisante la façon dont s’est déroulé le projet au cours des mois.

Nous allons maintenant étudié plus précisément les tâches imposés par ce planning.

Chapitre 2

Parser & Interface graphique

La parser est la partie du programme qui prend en entrée un fichier texte de description de scène 3D, dans lequel sont listés les différentes objets - sphères, sources de lumière, textures... - constitutifs de la scène finale. Avant de concevoir le parser en tant que tel, il fallait définir un langage de description de scène.

2.1 Format inital du fichier de description de scène

Dans un tout premier temps, nous avons utilisé un format très simplifié qui ne prévoit que la place des sources lumineuses, des objets - en précisant leurs coordonnées - ainsi que la place de la caméra.

0 -60 0 -230 250 0 0 45
1 60 0 -230 0 0 255 45
1 0 0 -340 255 255 0 45
1 0 -15 -225 255 255 255 10
2 -500 0 -100 255 255 255
2 0 0 -140 150 80 150
2 0 0 0 50 50 50

FIG. 2.1 – Fichier de description de scène primitif

Ce premier fichier de description de scène était "parsé" à l'aide d'un code en C mais il nous a très rapidement fallu trouver un autre moyen de description de scène afin de rendre ce fichier transparent pour l'utilisateur. Nous nous sommes alors interrogés sur l'existence d'outils permettant d'effectuer cette opération de manière structurée. Nous avons consulté les codes sources de POV afin voir comment ses concepteurs avaient abordé le problème. Ces derniers ont créé un parseur composé de deux parties : un *tokenizer* dont la fonction est d'extraire les expressions régulières du fichier de description de scène et de les passer au parser à proprement parler sous la forme de *token*. Le parser, quant à lui, gère les règles de grammaire pour pouvoir "remplir" la scène et la charger en mémoire.

2.2 Les différents objets

2.3 Format final du fichier de description de scène

Avant de se pencher sur la conception propre du parser, ils nous a fallu définir les différents éléments constitutifs d'une scène et les coder en C afin de pouvoir les utiliser simplement. Notre raytracer est capable

de “traiter” deux sortes d’objets géométriques : les sphères et les plans. Nous avons également besoin de sources de lumière, de matériaux ainsi que d’une “caméra” qui situe le point de vue de l’utilisateur vis-à-vis de la scène.

Comme nous codons en C, ces différents objets seront définis par des structures.

2.3.1 la scène

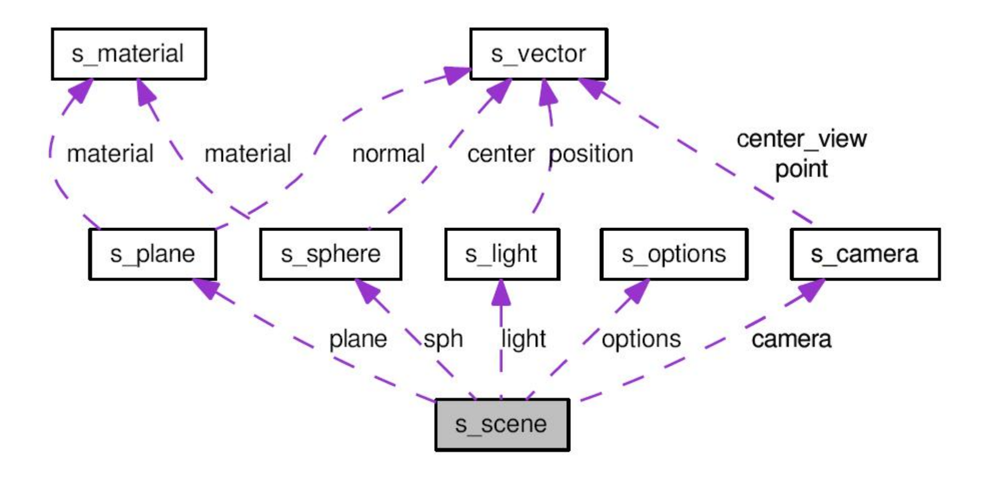


Fig. 2.2 – Graphe de collaboration de s_scene

Une scène est définit par :
- la résolution de l’image
- le nombre et sa liste de sphères, de plans, et de sources de lumières
- la position de la caméra et son point de fuite (vers lequel converge notre vision)

Une scène est dénit par : la résolution de l’image le nombre et sa liste de sphères, de plans, et de sources de lumières la position de la caméra et son point de fuite (vers lequel converge notre vision).

Les précisions sur les objets, leurs attributs, leurs coefficients sont consultables dans le manuel d’utilisateur.

2.3.2 Le fichier de description scène KFR

Les différents fichiers de description de scène portent l’extension *kfr*. Ils se présentent tous de la même manière, c’est-à-dire :

Scene{
Width = largeur de l'image générée (en pixels) ;
Height = hauteur de l'image générée ;
Materials = nombre de matériaux ;
Planes = nombre de plans ;
Spheres = nombre de sphères ;
Lights = nombre de sources lumineuses ;
Camera = coordonnées cartésiennes de la caméra ;
}
Material{
Id = 2 ;
Type = marbel ;
RGB = 0.0, 255.0, 255.0 ;
Reflection = 0.5 ;
Specular = 1.0 ;
Roughness = 60.0 ;
Perlin = 1.0 ;
Bump = 0.0 ;
}
Sphere{
Center = 440.0, 290.0, 0.0 ;
Radius = 100.0 ;
Material = 2 ;
}
Light{
Position = 0.0, 240.0, -100.0 ;
Intensity = 0.2, 0.2, 0.2 ;
}

FIG. 2.3 – format d'un fichier de description de scène 3D

Il est bien sûr possible de mettre autant de sphères, de matériaux ou de sources de lumière que l'utilisateur le désire. Des fichiers de description de scène complets, sont disponible dans le logiciel KF-Ray.

2.4 Le parser

Lex et Yacc sont deux outils d'aide pour la réalisation de programmes qui effectuent des transformations sur des entrées structurées.

Deux tâches reviennent immanquablement dans le cas des programmes acceptant des entrées structurées : faire apparaître des éléments significatifs dans le texte source, et trouver la relation entre ces éléments. Pour notre programme de recherche dans un texte, les éléments sont les lignes du texte, en distinguant les lignes où l'on a trouvé la chaîne cible des autres lignes. Les éléments sont les noms des variables, les constantes, les chaînes, la ponctuation... Cette séparation en éléments (ou *tokens*, comme on les dénomme habituellement) constitue l'analyse lexicale. Lex facilite cette opération en produisant à partir d'une liste de descriptions des tokens, une procédure en C pour les reconnaître, qui est appelée un analyseur lexical. L'ensemble des descriptions données à Lex est appelée une spécification lex.

Les tokens sont données à Lex sous forme d'expressions régulières. Lex convertit ces expressions régulières en une forme utilisable par l'analyseur lexical pour balayer le texte extrêmement vite, indépendemment du nombre d'expressions à rechercher. Un analyseur produit par Lex est à cooup sûr plus rapide que celui que l'on pourrait écrire nous-mêmes en langage C.

Au cours de la décomposition des entrées en tokens, le programme doit souvent établir les liens entre ces derniers. Cette tâche constitue l'analyse syntaxique, et la liste des règles qui régissent les relations que comprend le programe est appelé une grammaire. Yacc prend en entrée une description concise d'une grammaire et produit une procédure C, l'analyse syntaxique, qui est précisément capable d'effectuer l'analyse syntaxique

de cette grammaire, en repérant automatiquement qu'une séquence de tokens satisfait à une des règles de grammaire ou qu'il y a une erreur de syntaxe.

2.4.1 définition d'une expression régulière

Lorsque l'on écrit une spécification Lex, on créé un ensemble de formes auxquelles les entrées doivent correspondre. Il découpe les entrées en chaînes que nous appelons tokens. Lex ne produit aucun fichier exécutable, mais il traduit la spécification lex dans un fichier contenant une procédure C appelée yylex(). Une expression régulière est une description de forme faite à l'aide d'un métalangage. Les caractères de ce métalangage sont un sous-ensemble des caractères ASCII disponibles sous UNIX et MS-DOS.

.	désigne tout caractère isolé à l'exception du caractère de passage à la ligne
*	désigne l'expression qui le précède zéro fois ou davantage
[]	construit une classe lexicale désignant tout caractère cité entre les crochets. Si le premier caractère spécifié est l'accent circonflexe "ˆ", elle désigne tout caractère sauf ceux qui se trouvent entre les crochets. Un tiret indique un intervalle de caractères.
ˆ	désigne un début de ligne s'il figure en tête d'une expression régulière
$	correspond à une fin de ligne s'il figure en dernier caractère d'une expression régulière
\	caractère d'échappement des métacaractères
+	correspond à une ou plusieurs occurences de l'expression régulière qui précède
?	désigne zéro ou une occurrence de l'expression régulière qui le précède
/	désigne l'expression régulière qui précède, si elle est suivie de ce qui ets indiqué après le /
()	regroupe des expressions régulières pour en former d'autres plus vastes

Généralement, c'est à partir d'expressions régulières simples que sont construites des expressions régulières complexes.

Nous commençons par préciser ce qu'est une ligne blanche, que nous considérons comme étant toute ligne uniquement composée d'espaces et d'un caractère "\n". Les espaces à leur tour ne se composent que d'espaces blancs ou de tabulations.

L'expression régulière correspondante est donc :
ˆ[\t]*\n

Voici maintenant la description d'un commentaire :
ˆ[\t]*"/*".*"*/"[\t]*\n

Il s'agit d'un commentaire isolé, non imbriqué, et écrit sur une seule ligne, avec un texte quelconque à l'intérieur des "/*" et "*/". Nous avons dû mettre entre guillemets l'étoile et la barre oblique lorsqu'ils apparaissent dans le commentaire puisque ce sont des caractères spéciaux. Il faut cependant ajuster cette expression car elle n'accepte pas un commentaire ce de genre :
/* commentaire */ /* commentaire

Le commentaire se poursuit sur les lignes suivantes mais dans notre expression nous avons dû l'interdire par l'opérateur ".". Si ce dernier caractère était autorisé, il y aurait un risque de faire déborder le tampon interne de Lex avec un commentaire trop long.

Il est possible de contourner ce problème en ajoutant un mode de contexte COMMENTAIRE dans lequel nous entrons dès le début d'un commentaire. Nous ne retournerons au mode normal que lorsque la fin de commentaire sera atteinte. Ce contexte n'est pas utile pour un commentaire tenant sur une seule ligne. Nous avons bien codé ce mode CONTEXTE mais il s'est avéré qu'il ne fonctionnait pas correctement lors de plusieurs tests. Nous sommes alors resté sur le modèle du commentaire précédemment exposé.

Dans notre parser, il est nécessaire de pouvoir reconnaître des entiers ainsi que des nombres flottants. Voici donc les expressions régulières correspondantes :

- ?[0-9]+ expression régulière représentant un entier

- ?([0-9]*\.[a-z0-9]+) expression régulière désignant un nombre flottant

Nous devons à présent reconnaître un certain nombre de mots-clés apparaissant dans le fichier de description de scène :

ˆ{[Ss]cene}([\t\n]*\{) expression régulière qui reconnaît le mot "scene" ou "Scene" placé en début de ligne et suivi d'une accolade ouvrante, cette dernière pouvant lui être directement accolée ou lui être séparée par une tabulation ou un retour à la ligne

Les autres objets - sphère, plan, matériau et sources de lumière - sont décrits d'une manière analogue.

Les caractères ";", ",", "=" et "}" sont reconnus par Lex mais ne sont associés à aucune action : Lex les lit mais ne s'en préoccupe pas.

Et enfin, nous avons défini des expressions régulières pour chacun des mots que doit pouvoir reconnaître le parser :

ˆ{separator}*[Rr]eflection expression régulière qui reconnaît les mots "reflection" ou "Reflection" placé en début de ligne et précédé ou non d'un séparateur (espace ou tabulation)

Nous avons pu tester la validité de ces différentes expressions régulières avec un petit logiciel nommé "The regex Coach" :

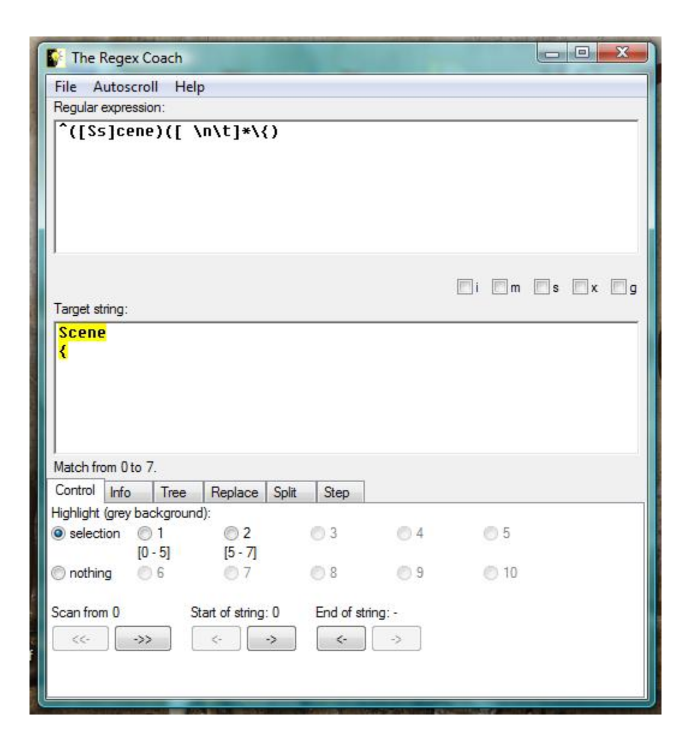


FIG. 2.4 – The Regex Coach

2.4.2 Yacc

Alors que Lex reconnaît les expressions régulières, Yacc reconnaît toute une grammaire. Lex décompose d'abord le flot d'entrée en morceaux, dits tokens ou unités syntaxiques, puis Yacc reprend ces morceaux et le regroupe ensemble logiquement. Il faut redéfinir dans le Yacc les token définis dans le Lex ainsi qu'une union qui permet de réunir les différents types de retour définis dans le Lex. Pour chaque règle de grammaire,

l'utilisateur peut associer du code C qui sera exécuté à chaque fois que la règle est reconnue dans les données d'entrée. Ces parties de code sont nommées actions et peuvent retourner des valeurs et récupérer des valeurs retournées par des actions précédentes. L'analyseur lexical peut aussi renvoyer des valeurs avec les token (la valeur entière ou flottante que représente le token qui définit les entiers ou les flottants). Voici des exemples d'actions attachées à leur règle de grammaire :

```
line_CAMERA : CAMERA FLOAT FLOAT FLOAT { camera0.point = CreateVector($2, $3, $4) ; }
line_ID : ID INTEGER { list_mat0[compteur_mat-1].id = $2 ; }
line_TYPE : TYPE NORMAL { list_mat0[compteur_mat-1].type = 0 ; }
        | TYPE TURBULENCE { list_mat0[compteur_mat-1].type = 1 ; }
        | TYPE MARBEL { list_mat0[compteur_mat-1].type = 2 ; }
        | TYPE WOOD { list_mat0[compteur_mat-1].type = 3 ; }
        | TYPE CHECKER { list_mat0[compteur_mat-1].type = 4 ; }
```

Fig. 2.5 – Actions Yacc

Pour faciliter la communication entre la grammaire, les actions et le parser, on utilise les symboles de retour $$,$2,$3... Pour obtenir des valeurs qui ont été retournées par des actions précédentes, l'action utilise les pseudos-variables $1,$2... qui réfèrent aux valeurs retournées par les symboles de droite d'une règle.

2.5 Glade 2 et GTK+ 2

2.5.1 Description

Dans le cahier des charges et des spécifications, nous avions convenu de l'allure que devrait avoir notre interface graphique.

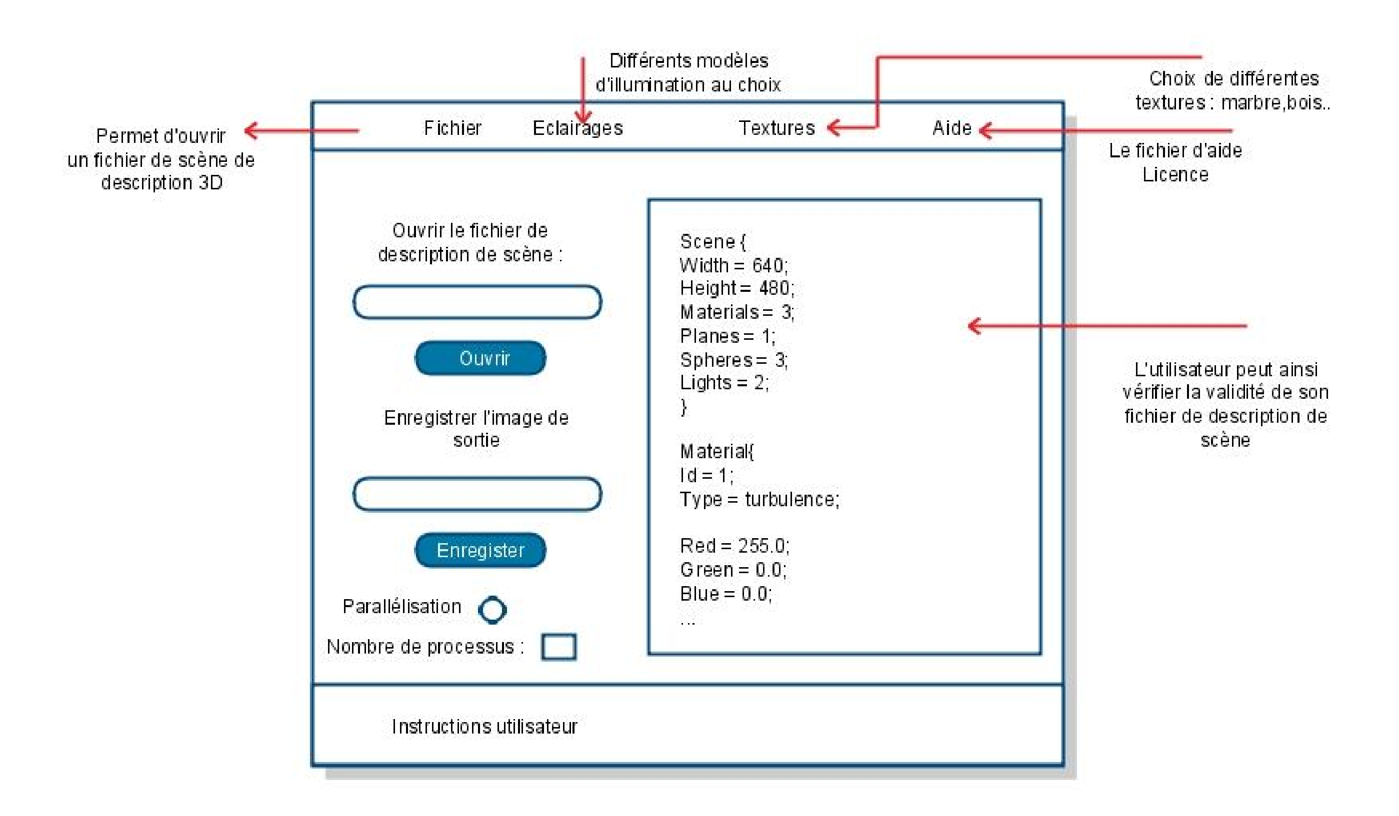


Fig. 2.6 – Interface schématique

L'interface du projet a été réalisée au moyen de la librairie graphique GTK+ (The Gimp Toolkit), librairie que nous avions déjà utilisée lors du projet court du premier semestre et à laquelle nous étions donc déjà sensibilisés. L'interface regroupe en quelques boutons l'ensemble des options disponibles dans le programme, qu'il s'agisse des textures ou du parallélisme. En outre, elle se doit d'être ergonomique, intuitive et ludique.

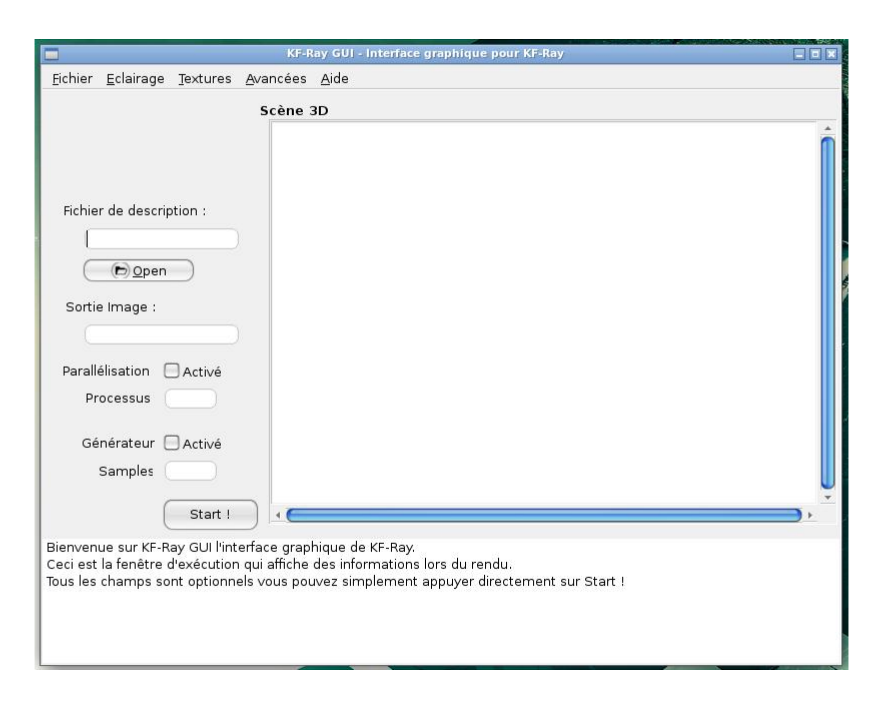


Fig. 2.7 – Frontend KF-Ray GUI

2.5.2 Réalisation

Nous avions prévu dans un premier temps d'obtenir un affichage dynamique via Glade. Seulement, nous avons rencontré un problème lors de l'implémentation du parallélisme. En effet, GTK refusait de se lancer lorsque l'on compilait avec MPI : apparemment, l'appel à gtk_main() ne pouvait se faire. Nous avons tenté de le déboger en utilisant Valgrind (c'est à cette occasion que nous nous somme aperçus que c'était l'appel à gtk_main() qui ne passaitpas). Nous avons tenté de contourner le problème par plusieurs moyens bien que nous n'en connaissions pas la source exacte : vérifier la compatibilité des librairies de Glade avec celles de MPI, utiliser des fork... Mais le problème a persisté avec toujours la même erreur, identifiée grâce à Valgrind.

Nous avons alors décidé de séparer complètement le frontend (GTK) du backend (le programme en tant que tel). Finalement, cette répartition s'est avérée être une bonne solution puisque l'on peut facilement accéder au programme en backend et le lancer en utilisant les options sur les lignes de commande, l'interface graphique n'étant qu'un "plus" , afin de rendre l'application accessible au plus grand nombre.

Par ailleurs, nous avons du prendre du recul face aux outils utilisés. Glade 2 est "obsolète" d'après les développeurs eux-mêmes dans la mesure où il génère lui-même du code C. Etant donné qu'il s'agissait d'un outil déjà présent sur les pc de l'école et que nous l'avions déjà utilisé lors d'un projet court, nous nous en sommes servis tout en gardant à l'esprit ses défauts. C'est pourquoi nous avons codé "à la main" le plus fréquemment possible les boîtes de dialogue, boutons,... afin de garder un maximum de contrôle sur le développement de l'interface. Glade reste très utile quant à disposer rapidement les différents éléments dans la fenêtre principale.

Chapitre 3

L'algorithme de Ray-tracing

3.1 Principe

En 1968, Arthur Appel publie un article sur la synthèse d'images qui pose les fondements de la technique du lancer de rayons (ray-tracing) que Turner Whitted popularisa en 1980. Depuis, ving-huit années de recherche fertiles en idées et en travaux ont considérablement enrichi l'informatique graphique aussi bien dans la recherche d'un plus grand réalisme que dans l'amélioration des performances des algorithmes à tel point que les images de synthèse sont parvenues au stade industriel et sont très largement utilisées, aussi bien pour le cinéma, la conception de pièces mécaniques, l'architecture, la médecine, les simulateurs (aviation, sport...), la visualisation scientifique (météorologie, géologie,dynamique des fluides...). De plus l'amélioration prévisible de la qualité des images par effets de rendus encore plus réalistes et la chute des temps de calculs permettent d'entrevoir des applications totalement nouvelles par l'introduction de l'animation en temps réel et l'intéraction avec un monde partiellement ou totalement numérique.

3.1.1 Le modèle de l'appareil photo simplifié

Au début des années 60, les informaticiens s'avisèrent que les modèles proposés en optique géométrique par les mathématiciens et les physiciens étaient d'excellents candidats pour la génération d'images de synthèse. A la base de l'idée du lancer de rayons, il y a un modèle d'appareil photo simplifié. Imaginons une boîte totalement fermée. Sur un côté de celle-ci, on plaque une feuille photosensible qui servira de pellicule et sur le côté opposé, on perce un minuscule trou (sténopé) avec une aiguille, de façon à ce que chaque point de l'espace à l'extérieur de la boîte ne puisse envoyer qu'un photon à travers ce trou. Le fait que le trou soit très petit évite d'avoir une pellicule surexposée : chaque point de celle-ci sera en rapport avec un unique point de l'espace (symbolisé sur la figure par un segment) :

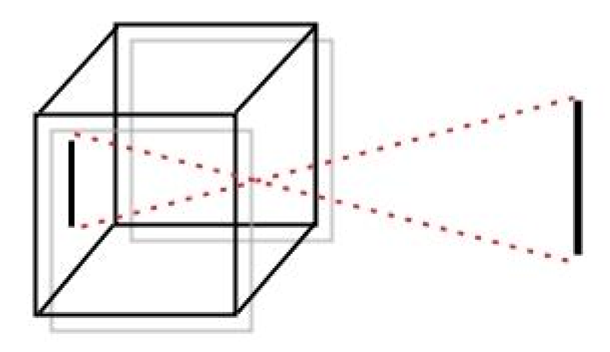


Fig. 3.1 – Modèle de l'appareil photo simplifié

3.1.2 L'algorithme

On place l'oeil de l'observateur au niveau du trou de l'appareil photo, et on interpose un écran entre l'oeil et l'image. Puisque l'écran est décomposé en pixels, on “lance” un rayon depuis l'oeil par chaque pixel et on “suit” ce rayon dans la scène. Cela revient à calculer l'équation d'une droite passant par l'oeil et un pixel, puis par calcul géométrique à déterminer son intersection avec les différents objets de la scène. S'il n'intersecte rien, le couleur du pixel est celle du “fond”. Sinon, on calcule la couleur de l'objet au point d'intersection. Cette couleur tient compte de l'ombre éventuelle (si la source lumineuse est cachée par un autre objet), de la couleur locale de l'objet et éventuellement des apports des autres objets de la scène (si l'objet est réfléchissant ou transparent). La détermination des ombres et des apports des autres objets nécessite de relancer d'autres rayons depuis le point d'intersection : vers les sources lumineuses pour l'ombre, grâce aux lois de Descartes pour la réflexion ou la transmission.

Le principe de l'algorithme est donc récursif. Chaque rayon donne naissance au rayon ombre et éventuellement naissance aux rayons réfléchis et transmis. Le calcul s'arrête quand un rayon se perd à l'infini ou rencontre un objet non spéculaire et non transparent.

Algorithme 3.1 Algorithme de ray tracing utilisé

Entrée : Scène 3D (caméra comprise)
Sortie : Image PPM

Début

Pour chaque ligne de balayage de l'image **Faire**
  Pour chaque pixel de la ligne de balayage **Faire**
    On lance un rayon du pixel vers le point de fuite (centre de projection)
    On initialise $profondeur \leftarrow 0$
    On initialise $\lambda \leftarrow INIT_\lambda$ $//INIT_\lambda$ est un nombre assez grand pour les objets les plus distants
  FPour
FPour

Pour chaque objet **Faire**
  Si il y a intersection entre l'objet courant et le rayon **Alors**
    Si cette intersection est plus proche que la précédente **Alors**
      On affecte à λ sa nouvelle valeur d'intersection
      On affecte à $mat_courant$ ←matériau courant de l'objet intersecté
    FSi
  FSi

Si il y a intersection **Alors**
  Pour chaque source de lumière **Faire**
    On envoie un rayon du point d'intersection à la source de lumière
    Si le point n'est pas dans l'ombre **Alors**
      Si $profondeur < profondeur_maximum$
        Si l'objet est réfléchissant
          On relance un rayon de réflexion
        FSi
        Si l'objet est transparent
          On relance un rayon de réfraction
        FSi
        On calcule la couleur du pixel grâce aux modèles d'éclairement
        $profondeur \leftarrow profondeur + 1$
      FSi
    FSi
  FPour
FSi

Fin

3.1.3 Quelques notions d'optique géométrique

3.1.3.1 l'optique géométrique

L'optique géométrique n'est pas une réduction simpliste de l'optique ondulatoire mais une approximation qui modélise parfaitement la majorité des phénomènes optiques observables dans la vie de tous les jours et fournit l'un des meilleurs outils mathématique à l'informaticien pour l'image de synthèse, ce qui n'est pas le cas pour l'optique ondulatoire qui est très difficile à utiliser en synthèse d'image.

L'optique géométrique s'applique à des milieux transparents, homogènes et isotropes pour des objets dont les dimensions sont grandes devant la longueur d'onde de la lumière que l'on considère. Les sources de lumières sont consiédérées comme ponctuelles.

3.1.3.2 définitions et vocabulaire

couleur un rayon lumineux est un ensemble complexe de longueurs d'onde. Dans la suite, on supposera qu'il est toujours possible de décomposer cet ensemble et de se ramener à un ensemble équivalent de trois composantes : rouge, vert et bleu dont chaque composante est un pourcentage compris entre 0% (absence totale de la composante) et 100%. Le modèle RVB sera plus explicitement présenté dans le chapitre consacré au calcul de la couleur d'un rayon.

diffusion réémission d'une couleur par un objet suite à une intéraction avec son milieu.

indice de réfraction rapport de la vitesse de la lumière dans le vide sur la vitesse de la lumière dans un milieu transparent et isotrope considéré ($n = c/v$).

dioptre surface de séparation de deux milieux d'indices de réfraction différents.

plan d'incidence plan défini apr le vecteur directeur par le rayon d'incidence et la normale à la surface où aboutit ce rayon.

réflectance coefficient exprimant le pourcentage de couleur (RVB) réfléchi par la surface de séparation d'un dioptre.

réflection diffuse couleur réémise après absorption d'un ensemble de photons sur un ensemble d'atomes.

réflection spéculaire couleur réémise après rebond mais sans absorption d'un ensemble de photons sur un ensemble d'atomes.

réfraction passage d'un rayon à travers un dioptre selon les lois de Snell-Descartes.

réfringence propriété de réfracter la lumière.

transmission certains corps laissent passer une partie de la couleur d'un rayon incident à travers la surface de séparation d'un dioptre. On dit que la lumière est transmise à travers cette surface.

transmission diffuse si la lumière intéragit avec le milieu qu'elle traverse par une suite d'absorptions et de réémissions, on parle de transmission diffuse.

transmission spéculaire si la lumière interagit avec le milieu qu'elle traverse sans être absorbée, on parle de transmission spéculaire.

transmittance pourcentage de couleur (RVB) passant à la surface de séparation d'un dioptre.

3.1.3.3 les lois de l'optique géométrique

– les lois de Snell-Descartes

Loi de réflexion

1. le rayon réfléchi est dans le plan d'incidence
2. l'angle de réflexion est égal à l'angle d'incidence

Loi de transmission

1. le rayon incident et le rayon réfracté sont dans le plan d'incidence, ce qui signifie d'un point de vue mathématique : $i_1 = i_2$ et si $\overrightarrow{I} = -\overrightarrow{Rayon\,incident}$ alors $\overrightarrow{R} = 2 * \overrightarrow{N} * cos(i_1) - \overrightarrow{I}$.
2. les angles d'incidence et de réflexion sont liés par la relation suivante : $n_1 sin(i_1) = n_2 sin(i_2)$.

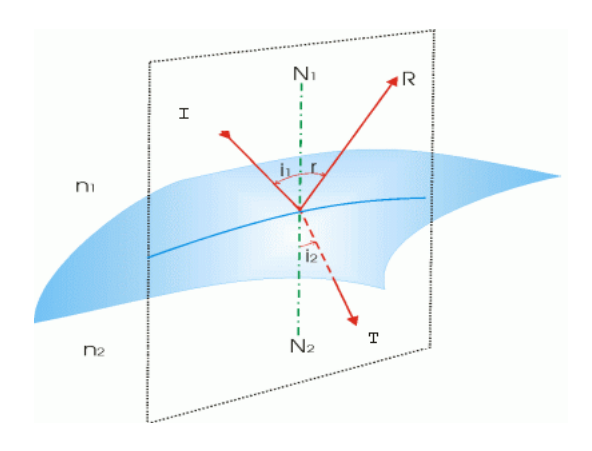


Fig. 3.2 – Lois de Snell-Descartes

Effets non (ou difficilement) traités en optique géométrique :

Certains effets liés à la nature ondulatoire de la lumière ne peuvent être restitués en optique géométrique et sont donc difficilement modélisables par le biais du ray-tracing.

les interférences lumineuses un rayon ne décrivant pas en lui-même les paramètres d'amplitude et de déphasage de l'onde, il n'est pas possible de reconstiuter des franges d'interférence.

la diffraction elle est liée à la longueur d'onde et provoque un phénomène d'interférence. Or l'optique géométrique se place explicitement dans des situations où la dimension des objets est grande devant les longueurs d'onde, alors que les phénomènes de diffraction ont lieu uniquement dans les cas où la dimension des objets est comparable à celle des longueurs d'onde.

3.2 Ray Traycer

3.2.1 Premiers pas

Dans un premier temps, nous avons réalisé un petit raycaster, c'est-à-dire un raytracer simplifié qui ne gère pas la récursivité du lancer de rayon : il ne peut donc pas y avoir ni réflexion ni réfraction. L'objet le plus simple à modéliser est une sphère. En effet, en coordonnées cartésiennes, il est aisé de déterminer l'équation d'une sphère puis de déterminer le ou les points d'intersection entre une demi-droite (notre rayon en l'occurrence) et cette sphère. Cette première approche s'est effectuée à l'aide de GTK, à une époque où nous n'avions pas encore implémenté le parallélisme et où GTK constituait notre interface principale. Cette librairie possède un composant nommé *gdkRGB* qui représente un buffer de couleur RVB.

Les images obtenues au début de notre programmation n'étaient que des cercles 2D afin de bien maîtriser les pixmap avant de tenter de passer à la 3D. Afin d'obtenir cet effet, nous nous sommes penchés sur la création de sources lumineuses ponctuelles.

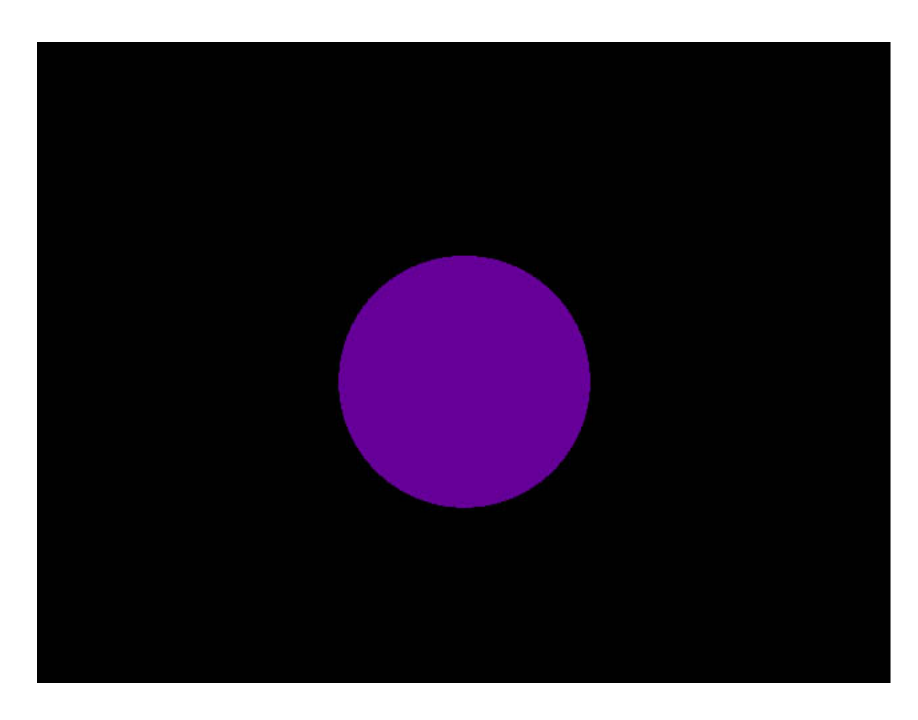


Fig. 3.3 – Sphère avec un raycast sans éclairement

3.2.2 Les rayons

3.2.2.1 Rayon initial

La caméra représente le point à partir duquel l'observateur regarde la scène. Le rayon initial est calculé pour le pixel (x,y).

3.2.2.2 Rayon réfléchi

La réflexion d'un rayon sur une surface est basée sur les lois de Descartes. Il s'agit de faire partir un nouveau rayon à partir du point d'intersection du rayon incident. L'angle de départ du rayon réfléchi est le même que celui du rayon incident (toujours d'après les lois de Descartes). On calcule alors le vecteur normal au point d'intersection et l'on obtient le vecteur réfléchi à l'aide de la formule suivante : $\overrightarrow{R} = 2 * \overrightarrow{N} * cos(i_1) - \overrightarrow{I}$.

3.2.2.3 Rayon réfracté

La réfraction a lieu quand un objet est transparent et que son indice de réfraction est différent de celui du milieu d'où provient le rayon. Ce dernier passe alors à travers l'objet mais avec un certain angle. Si l'indice de réfraction est le même, le rayon est transmis, il n'y a pas de modification d'angle. En revanche, si ces deux indices ne sont pas identiques, et qu'il existe une relation particulière entre ceux-ci, il se peut même que le rayon soit "piégé" dans l'objet. On parle alors de réflection totale interne. La rayon réfracté est calculé à partir de la formule suivante (qui provient des lois des Descartes) : $\overrightarrow{T} = \left[n(\overrightarrow{N}.\overrightarrow{I}) - \sqrt{1 - n^2(1 - (\overrightarrow{N}.\overrightarrow{I}))^2} \right] \overrightarrow{N} - n\overrightarrow{I}$.

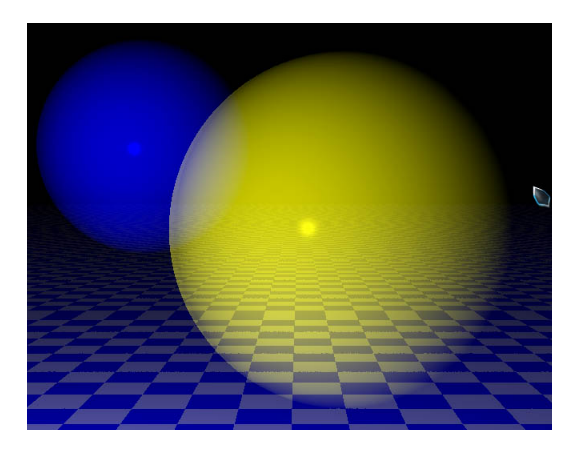


Fig. 3.4 – Transparence

3.2.2.4 Intersection rayon/objet

Nous n'avons implémenté que deux objets dans notre raytracer : les sphères ainsi que les plans. En effet, des deux figures sont relativement faciles à décrire en coordonnées cartésiennes.

3.2.3 Modèles d'illumination

3.2.3.1 le modèle RVB

Ce modèle utilise un système tridimensionnel orthonormé et définit un cube unité comme indiqué figure . Chaque axe correspond à une couleur primaire : rouge, vert, bleu. Une couleur est donc spécifiée en indiquant les contributions de chaque couleur primaire additive.

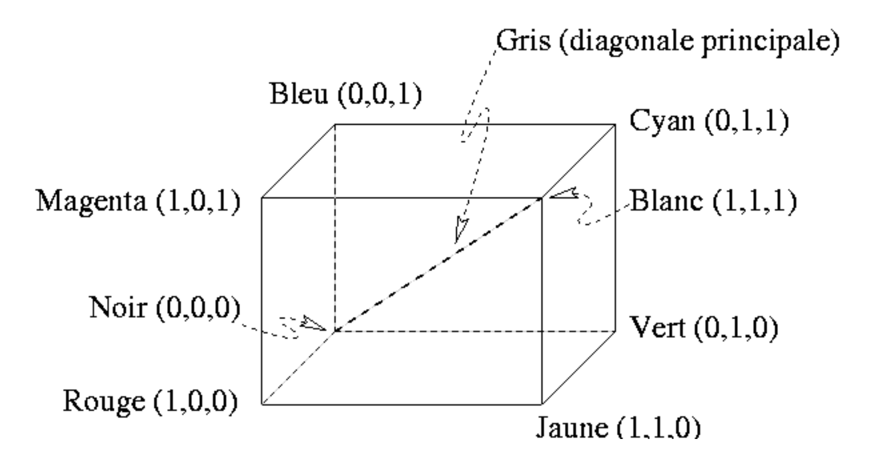


Fig. 3.5 – L'espace RVB

Ce modèle est relativement facile à utiliser et correspond aux dispositifs électroniques équipant les moniteurs couleur.

3.2.3.2 les sources de lumière

– La lumière ambiante : c'est une lumière qui éclaire toute la scène uniformément et qui est uniquement caractérisée par son intensité.
– les sources ponctuelles : ce sont des sources de lumière, supposées placées en un point précis et qui rayonnent la lumière radialement, elles sont donc caractérisées par leur intensité et leur position. Elles peuvent être isotropique ou anisotropique.
– les sources directionnelles : ce sont des sources de lumière, supposées à l'infini et qui éclairent la scène avec des rayons parallèles à une direction donnée ; elles sont donc caractérisées par leur intensité et leur direction.
– les sources de type Projecteur ou spot : ce sont des sources de lumière caractérisées par leur position, leur direction et un facteur de concentration.

3.2.3.3 définition d'un modèle d'éclairement

– Lumière émise : Les objets ne sont pas intrinsèquement émetteurs de lumière et n'éclairent donc pas les autres objets mais possèdent ce niveau minimum d'éclairement.
– Lumière ambiante : correspond au modèle le plus simple. On considère qu'il existe une source lumineuse présente partout et qui éclaire de manière égale dans toutes les directions. Ce modèle de lumière correspond au niveau minimum d'éclairage qui sera appliqué sur les objets. En terme physique cela correspond un peu au soleil réfléchi par tout l'environnement qui donne une sorte de lumière présente partout.

On définie l'intensité de cette lumière sur une surface comme suit :$Ip = pa * Ia$. Cette intensité lumineuse est constante sur toute la surface.

Ia désigne l'intensité de la lumière

pa est le coefficient de réflexion de la lumière ambiante par la surface

Ip correspond à l'intensité de la lumière résultant de la réflexion sur la surface.

3.2.3.4 modèles d'éclairement

1. **La réflexion diffuse (Modèle de Lambert)** : dans ce modèle, l'intensité en un point d'une surface dépend de l'angle formé entre le rayon de lumière qui touche le point de la surface et la normale à la surface. Plus l'angle formé entre le rayon de lumière et la normale au plan est faible, plus l'intensité lumineuse réfléchie visible par l'observateur est forte. Le principe physique qui se cache derrière ce modèle est simple. Notre source lumineuse émet une certaine énergie au mètre carré. Suivant l'incidence des rayons lumineux, cette énergie sera répartie sur une plus ou moins importante surface de l'objet. Si les rayons et la surface sont perpendiculaires, alors l'énergie lumineuse sera répartie sur la plus petite surface possible et donc l'énergie par unité de surface sera maximale.



Fig. 3.6 – Modèle de la réflexion diffuse

Si on considère que la surface est une surface Lambertienne (surface mate), alors, on peut considérer que la lumière qui arrive sur la surface est réfléchie dans toutes les directions de manière égale. Dans ce cas, la position de l'observateur ne compte pas. La lumière émise en direction de l'observateur dépend donc de :
• L'intensité de la source lumineuse Il
• L'angle θ formé par le rayon de lumière et la normale au plan
• Coefficient de réflexion pd de la lumière diffuse par la surface
• $0 \leqq pd \leqq 1$
• On obtient la formule :$Ip = pd.Il.cos(\theta)$

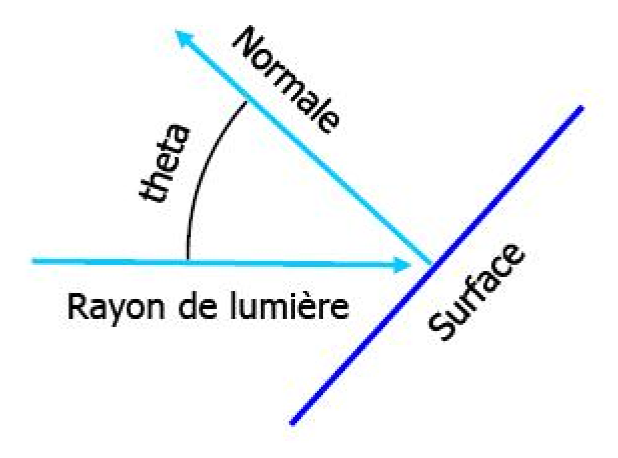


Fig. 3.7 – Illustration de la réflexion

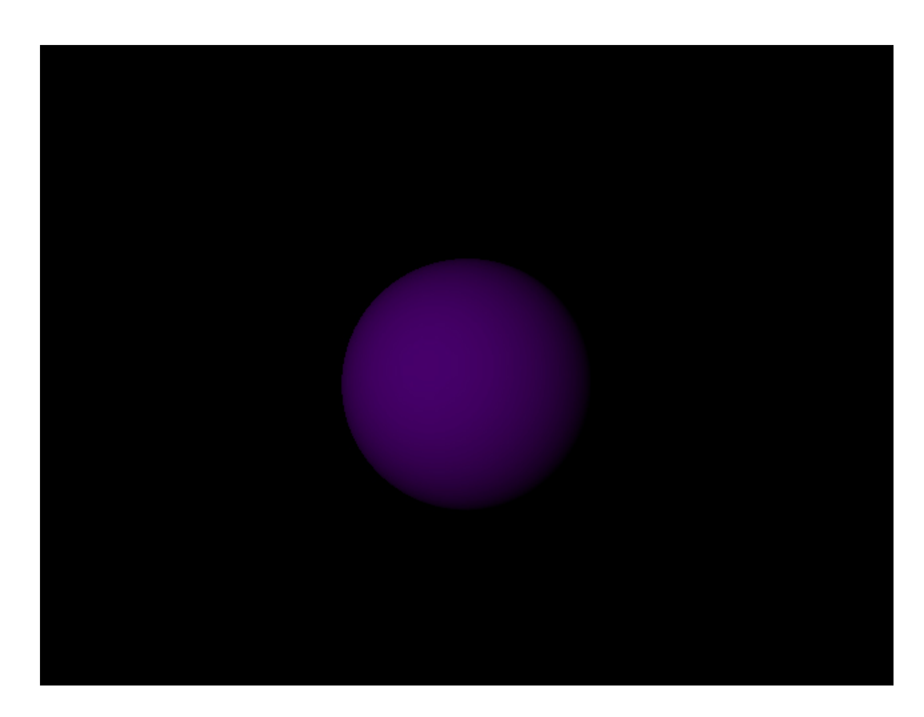


Fig. 3.8 – Sphère et modèle de Lambert

2. **La réflexion spéculaire (Modèle de Phong [1973])** : Le modèle de réflexion spéculaire se différencie du modèle de diffusion en faisant intervenir le point d'observation. Dans ce modèle les rayons de lumière sont réfléchis par symétrie par rapport à la normale à la surface. Ce modèle correspond aux propriétés de "miroir" des objets. Il faut calculer quel est le rayon réfléchi sur la face.

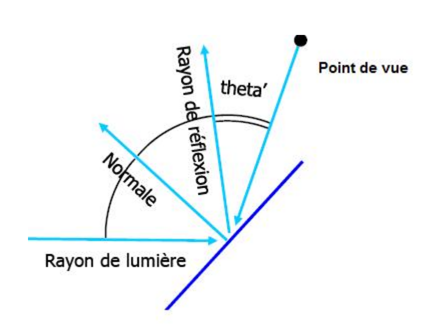


Fig. 3.9 – Modèle de Phong

Ensuite, l'intensité de la lumière observée dépend de :
• θ qui correspond à l'angle entre le rayon réfléchi et le point d'observation
• Il l'intensité de la source de lumière
• ps : coefficient de réflexion de la lumière spéculaire par la surface
• $0 \leqq ps \leqq 1$
• Si le rayon réfléchit vient directement dans l'oeil alors on a l'intensité maximum. Autour de cela, on peut quand même voir quelque chose d'un peu atténué. Cela veut dire que la surface ne réfléchit pas directement la rayon mais qu'il y a une certaine "diffusion" autour du rayon réfléchi. La fonction cosinus joue bien son rôle ici mais comme on veut pouvoir régler la diffusion autour du rayon réfléchi, on introduit le coefficient n. On obtient la formule : $Is = ps * Il * cos(\theta')^n$

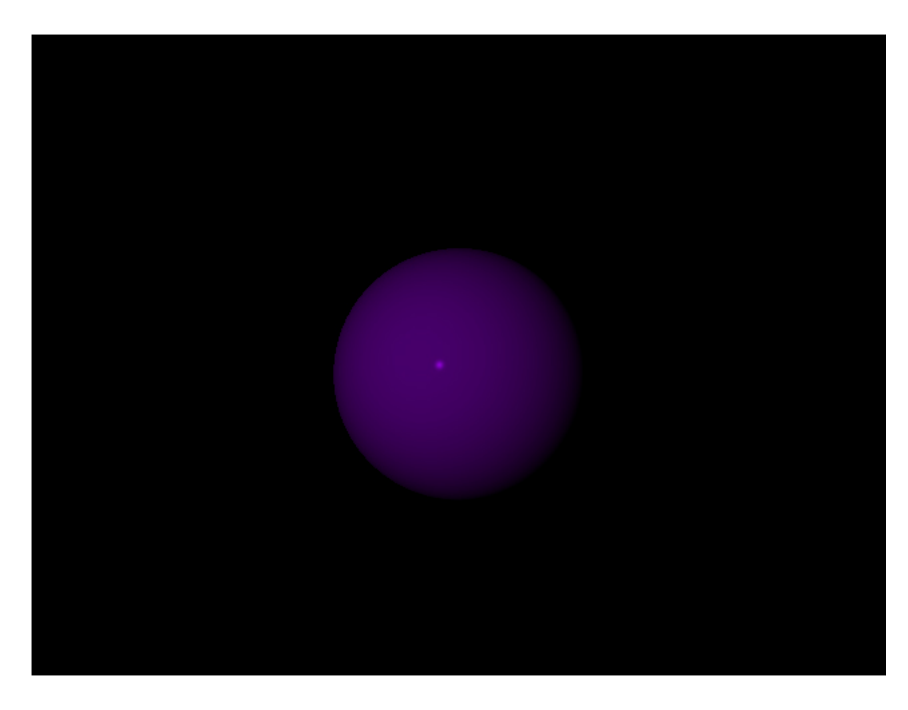


Fig. 3.10 – Sphère et modèle de Phong

Avec n la rugosité
• ∞ pour un miroir
• 1 pour une surface très rugueuse.
La rugosité corresponds à la brillance (shininess) : cette valeur détermine la taille et l'intensité de la tâche de réflexion spéculaire. Plus la valeur est grande, et plus la taille est petite et l'intensité importante.

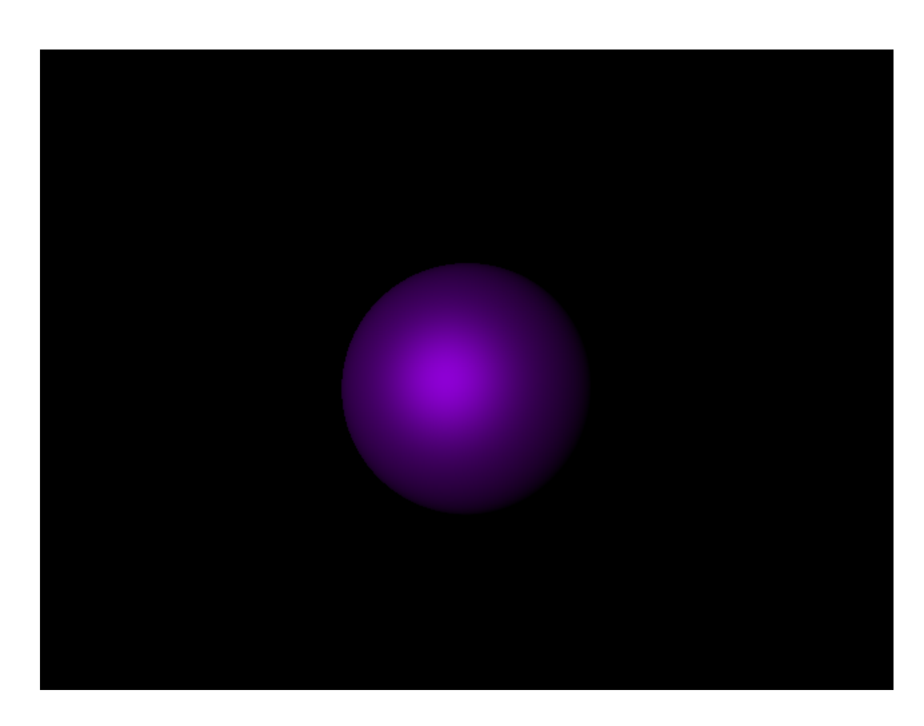


FIG. 3.11 – Effet de la rugosité pour une valeur petite de n

3. **La réflexion spéculaire (Modèle Blinn-Phong [1977])** : L'évaluation du rayon réfléchi est relativement coûteuse, il est donc nécessaire de développer une version simplifiée/accélérée de l'éclairage de Phong : il s'agit du modèle Blinn-Phong. Il repose sur l'évaluation du vecteur H, c'est-à-dire le vecteur séparant point de vue et rayon de lumière.

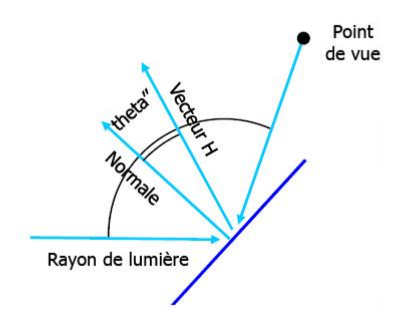


FIG. 3.12 – Modèle géométrique de Blinn-Phong

On obtient alors la formule suivante : $Is = ps * Il * cos(\theta")^n$, $H = \frac{V+L}{2}$ avec V : direction de vision et L : source d'éclairage.

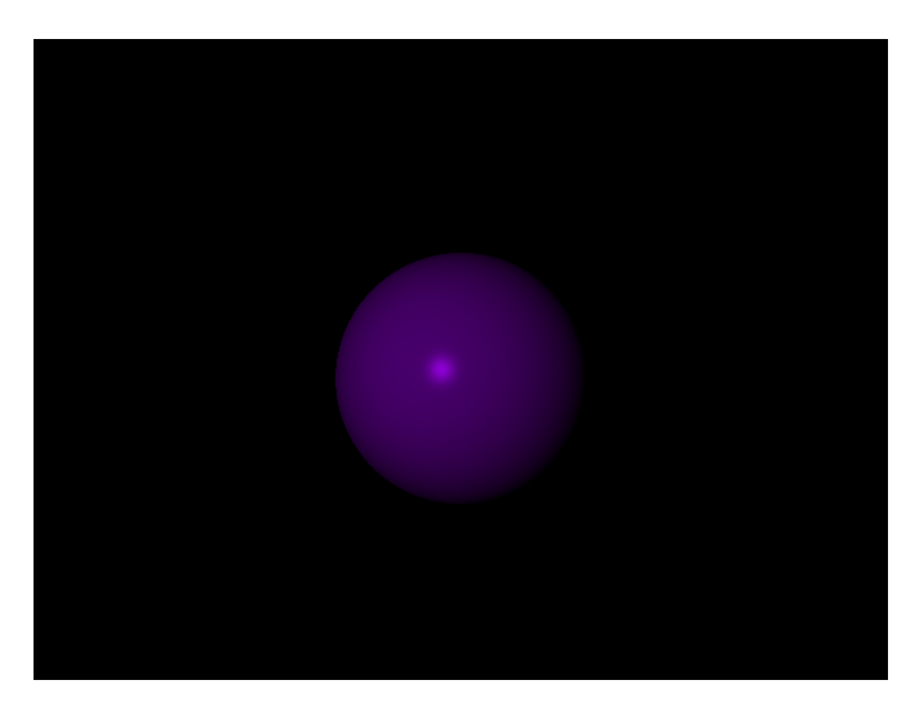


Fig. 3.13 – Sphère et modèle de Blinn-Phong

Nous obtenons des résultats très satisfaisants et qui correspond au modèle théorique stipulé en changeant également les coefficient de spécularité ou de rugosité dans les scènes 3 et 4 qui correspondent au figures proposées.

Nous laissons à l'utilisateur le choix d'utiliser le modèle et les coefficients qu'il souhaite facilement, correspondant à l'image qu'il souhaite.

3.2.4 Textures

La présence de textures dans les images de synthèse est certainement un des éléments déterminants pour obtenir des images réalistes. En effet, outre des détails de couleur, les textures peuvent apporter à une surface des détails de formes qui ne sont pas toujours aisées à obtenir par une modélisation géométrique directe.

3.2.4.1 texture procédurale

Afin de modéliser des motifs complexes (nuages, marbres . . .), il est nécessaire d'avoir à disposition des algorithmes capables de les générer de façon automatique. Pour introduire une grande diversité et par ailleurs un meilleur réalisme, ces algorithmes se doivent d'introduire de l'aléatoire dans ces motifs. C'est là qu'intervient le bruit de Perlin.

Ken Perlin est chercheur au ■ Department of Computer Science ■ au sein de l'université de New York. Il est aussi le directeur du ■ Media Research Laboratory ■ . Récompensées à de nombreuses reprises, ces recherches portent principalement sur l'animation, le graphisme et le multimédia.

Ce bruit est qualifié de cohérent et c'est ici que réside tout l'intérêt de son utilisation. En effet, il ne s'agit pas de générer des motifs totalement aléatoires, car sinon nous obtiendrions un motif "tacheté" très peu réaliste. En réalité, le bruit de Perlin est une somme de signaux aléatoires de fréquences et d'amplitudes différentes sachant que plus un signal est de fréquence élevée, plus son amplitude est faible.

Dans une image, les basses fréquences constituent le squelette alors que les hautes fréquences représentent plutôt les détails. Dès lors, il semble tout à fait normal d'associer les amplitudes élevées aux signaux de basse fréquence.

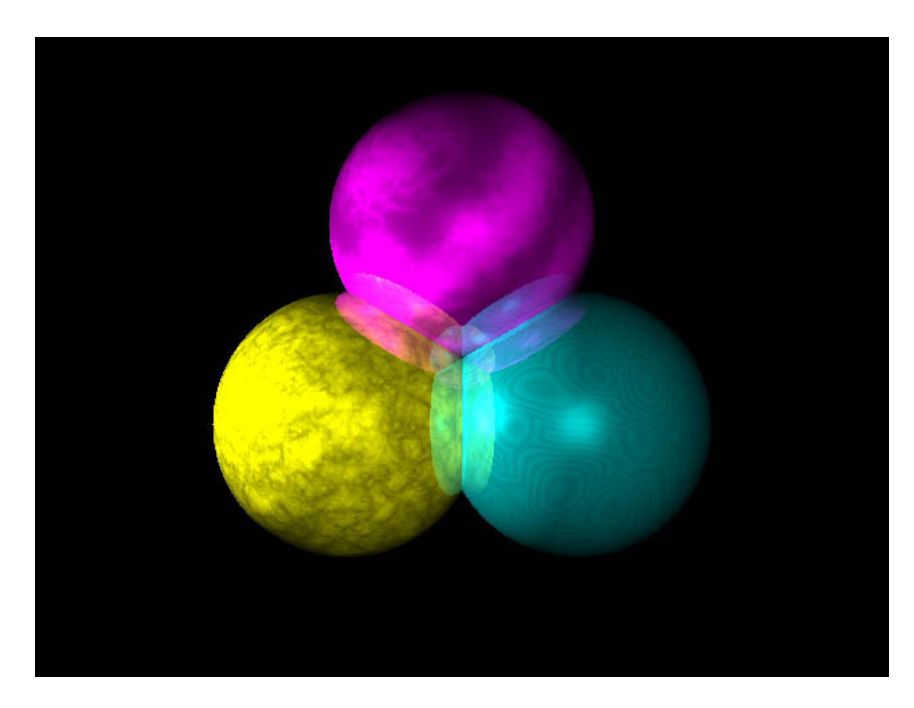


Fig. 3.14 – Illustrations des textures procédurales (turbulence, marbre, wood)

3.2.4.2 bump mapping

Cette technique permet d'ajouter à un objet à priori lisse un aspect rugueux. Pour se faire, nous allons perturber les normales des objets et donc, comme les normales rentrent en compte dans la plupart des calculs de luminosité, l'éclairage à la surface des objets. Ainsi deux points très proches pourront présenter un contraste très fort et ainsi créer un relief en surface.

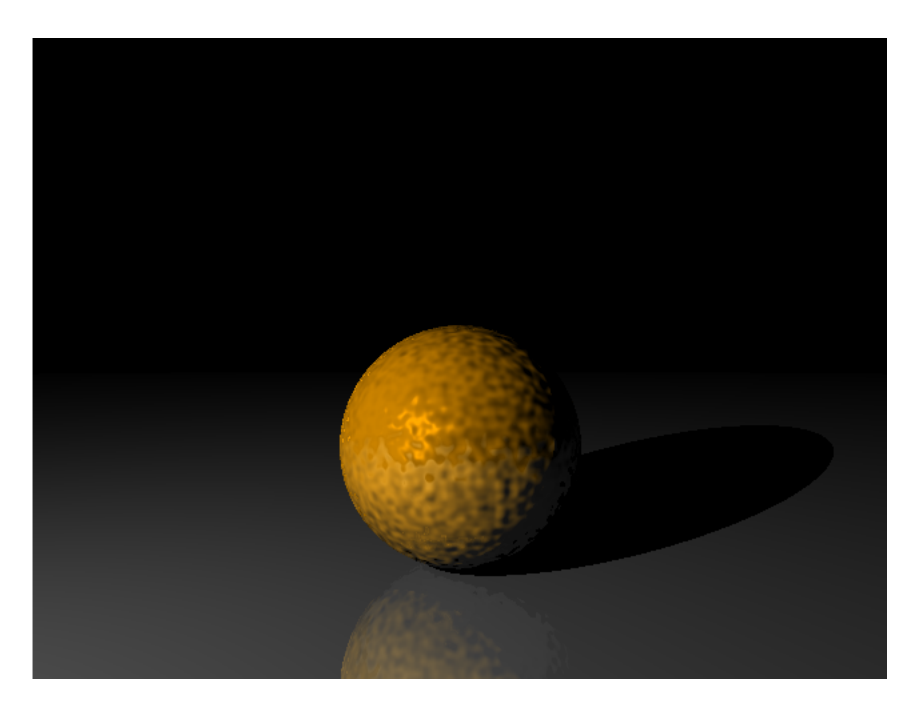


Fig. 3.15 – Effet du bump mapping

3.2.5 Anti-aliasing

Afin de lutter contre les phénomènes d'aliassage (dus à un échantillonage insuffisant de l'image) se manifestant sous les formes suivantes :
- présence de marches d'escaliers sur les contours,
- présence de moirés sur les textures (damier entre autre)
- petits objets entièrement ou partiellement cachés

La méthode de raytracing reposant sur l'utilisation de rayons de diamètre nul, il se crée sur les bords des objets un phénomène d'aliasing peu réaliste. Pour éviter cet effet de 'marches d'escalier', il existe plusieurs méthodes.

Voici celle que nous avons implémentée : au lieu de lancer un seul rayon par pixel, qui prendra pour intensité la valeur de l'illumination du rayon passant par son centre, on lance plusieurs rayons (en l'occurrence quatre) par pixel, uniformément répartis sur la surface du pixel. La valeur finale de l'illumination est la moyenne des valeurs de chaque rayon. Cette méthode donne de bons résultats, mais multiplie les temps de calcul par le nombre de rayons que l'on décide de lancer. La plupart des pixels n'ayant pas besoin d'être antialiasés, cette méthode est lourde.

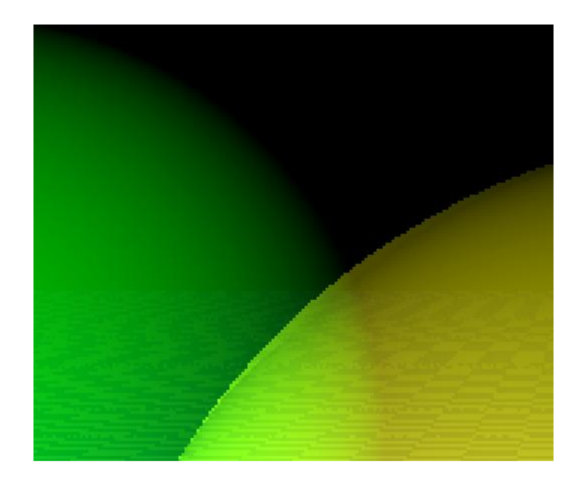


Fig. 3.16 – Observation du phénomène d'aliassage

Puisqu'il est inutile d'effectuer de l'antialiasing pour tous les pixels, on pourrait ne lancer qu'un seul rayon par pixel. Par contre, la valeur de l'intensité trouvée pour ce pixel serait comparée à celle du pixel à gauche et du pixel au dessus. Si un écart trop important existait entre ces trois intensités, on lancerait un nouveau rayon au hasard dans la surface du pixel, et l'on moyennerait avec la valeur précédente. Si la nouvelle valeur du pixel était encore trop éloignée de la valeur des deux voisins, on continuerait de lancer des rayons au hasard dans le pixel et de moyenner avec la valeur précédente, jusqu'à arriver soit à la profondeur maximale autorisée, soit jusqu'à obtenir une valeur conforme au paramètre du fichier de description de scène. Cette méthode n'est pas infaillible, mais permet dans une large majorité de cas d'obtenir un résultat acceptable, et possède l'avantage de ne pas engendrer de calculs inutiles ralentissant l'exécution du programme.

Certains algorithmes sont encore plus efficaces dont notamment celui de Monte-Carlo, basé sur un lancer stochastique de rayons.

3.2.6 La caméra

Au début, on lançait (pour des raisons pratiques) les rayons tous parallèles entre eux. Mais un problème est survenu lorsque nous avons voulu développer le motif du damier : on ne voyait pas correctement le damier sur le plan alors qu'on le voyait bien réfléchi sur une sphère situé au-dessus de celui-ci. Ce qui signifiait que le problème ne provenait pas de l'algorithme générant le damier. Après bien des hypothèses, nous avons déterminé la source du problème : les rayons étant toujours parallèles, il n'y avait aucun effet de perspective possible. Nous avons alors créé un "point de fuite" vers lequel convergent tous les rayons lancés. Il existe des solutions plus classiques de gestion de perspective mais notre solution a été trouvé intuitivement et reste simple tout en donnant des résultats convenables.

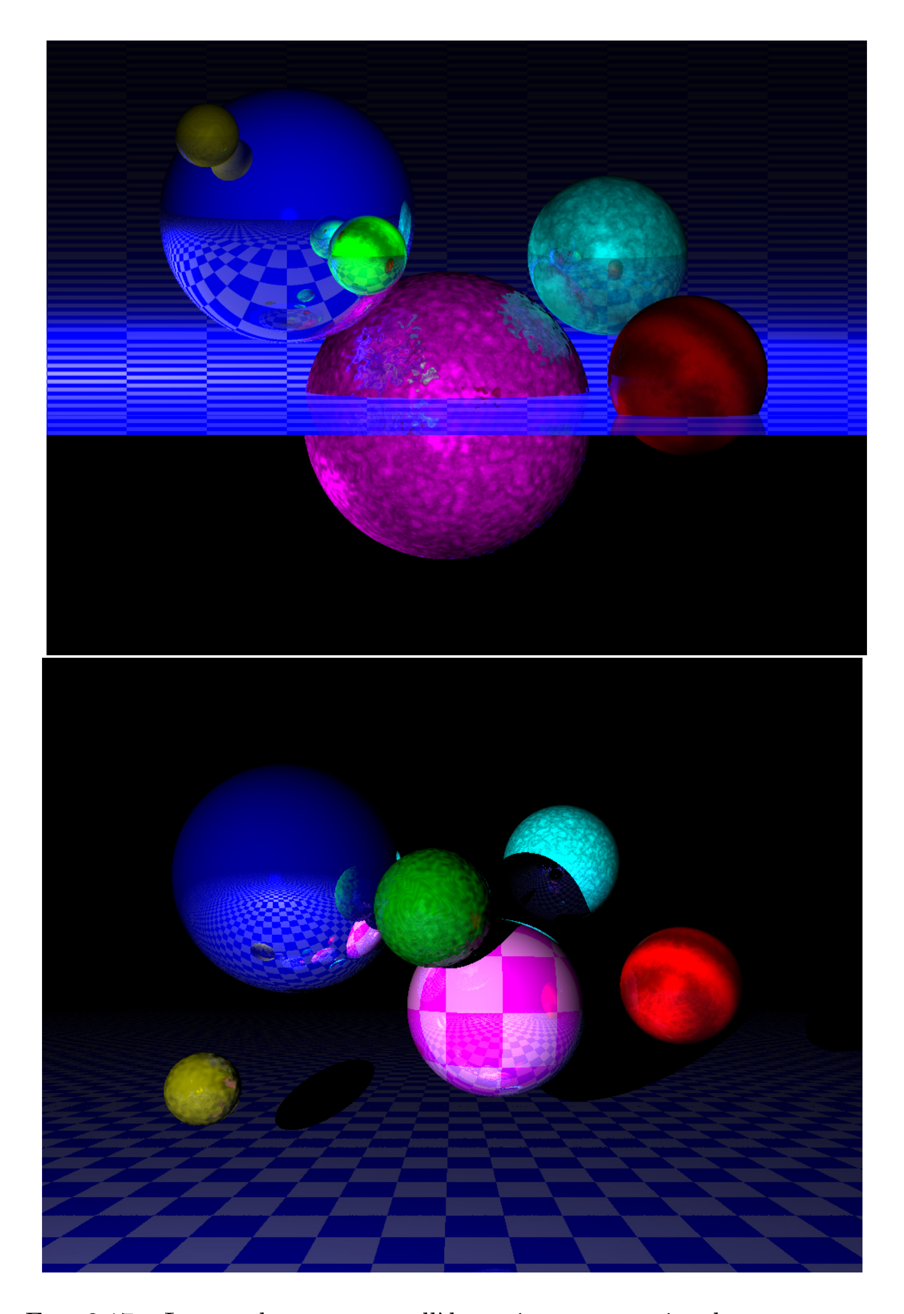


Fig. 3.17 – Lancer de rayons parallèles puis vers un point de convergence

3.2.7 Recul sur le rendu

Notre raytracer est capable de générer des scènes compliqués en un temps relativement moindre. Nous avons comparé notre scène2.kfr des trois sphères sur le même plan avec une scène presque similaire sur POV-Ray. Le rendu est quasi-similaire au détail près que les coefficients de spécularite, de rugosité ne sont pas les mêmes provoquent un rendu tout de même différent.

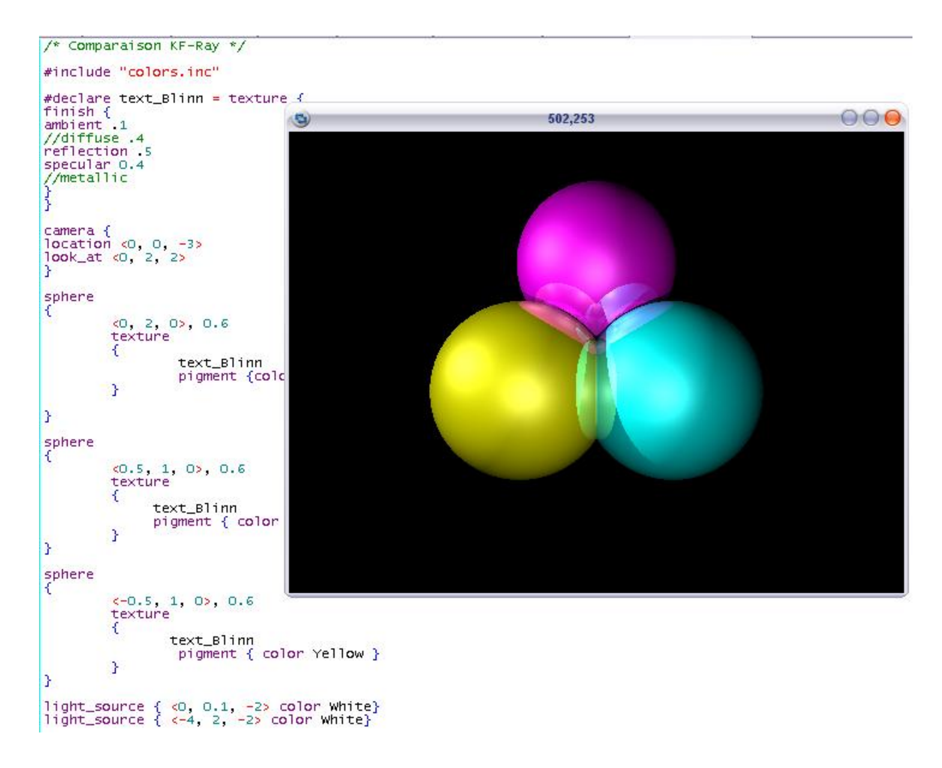


Fig. 3.18 – Scène similaire sur POV-Ray avec fichier de description de scène

Il est à noter qu'au niveau des performances, POV-Ray génère cette image sur la même machine en un peu plus d'une seconde tandis que KF-Ray génère cette image en un peu moins de deux secondes. Notre programme est capable de tenir la route face aux nombreux algorithmes d'optimisation de POV-Ray. Par le biais du parallélisme, KF-Ray peut se montrer beaucoup plus performant.

Chapitre 4

Parallélisme

4.1 Première approche

4.1.1 Principe

Il existe trois domaines sur lesquels on peut travailler afin d'optimiser le déroulement d'un algorithme :

1. la modélisation du problème c'est-à-dire les aspects physiques et sur sa mise en équations, aspects mathématiques ;
2. les méthodes numériques utilisées ;
3. sur le code lui-même : vectorisation, parallélisation par directives...

Le troisième point ets réservée à l'optimisation locale du code sur le processeur (vectoriel ou super scalaire) et sur le noeud de calcul (souvent multi processeur à mémoire partagée). Une approche générale et relativement simple à mettre en oeuvre, nous est donnée par le modèle SPMD pour "*Single Program Multiple Data*", où le même exécutable traîte sur chaque noeud de la machine parallèle une partie des données du problème.

Bien sûr, ces différents exécutables doivent se synchroniser et communiquer entre eux, ce qui est assuré en principe par l'utilisation d'une bibliothèque qui permet la mise en oeuvre du modèle de programmation par échanges de messages, le plus souvent MPI[1] (ou PVM[2]). Il faut alors répartir les données du problème et la première idée est de couper le domaine de calcul.

Par ailleurs, il est primordial de pouvoir évaluer les performances de l'algorithme parallélisé par rapport à sa version séquentielle. Le but réside dans l'étude du passage à l'échelle (scalability) de l'algorithme parallèle.

- $T_1(n)$: temps nécessaire à l'exécution du meilleur algorithme séquentiel pour résoudre une instance de problème de taille n avec un processeur.
- $T_p(n)$: temps nécessaire à l'exécution de l'algorithme parallèle considéré pour résoudre une instance de problème de taille n avec p processeurs.

On appelle alors accélération la quantité : $S(n,p) = \frac{T_1(n)}{T_2(n)}$.

On appelle efficacité la quantité : $E(n,p) = \frac{S(n,p)}{p}$.

[1] Message Passing Interface
[2] Parallel Virtual Machine

4.1.2 L'art de découpage une image

Il convient de choisir un grain optimal autrement dit, un découpage idéal. Il permet de répartir harmonieusement la charge de calcul sur les processeurs pour minimiser le délai total d'exécution. Les critères retenus pour déterminer ce découpage peuvent être plus complexes que simplement le nombre de noeuds du sous-domaine (par exemple la quantité de calculs à effectuer pour chaque élément du maillage peut différer d'une région à une autre..). Il doit en outre minimiser le surcoût engendré par ce découpage c'est-à-dire le volume des communications, relié directement à la taille des "interfaces" entre sous-domaines.

Nous avons après de nombreux essais fixés la granulité à 10 lignes par processus. Bien entendu, cela reste modifiable en fonction de la taille de l'image, nous laissons à l'utilisateur la possibilité de changer cette granulité grâce à l'option -l de KF-Ray.

Prenons un simple domaine rectangulaire maillé de façon régulière (h*w points) que nous découpons en parties "égales" destinées à être traitées par p processus. Etant donné que l'algorithme ne s'exécute que sur un pixel, sans se "préoccuper" des valeurs des pixels l'environnant, nous n'avons pas rencontré de problème de surcoût lié aux communications à la frontière des interfaces.

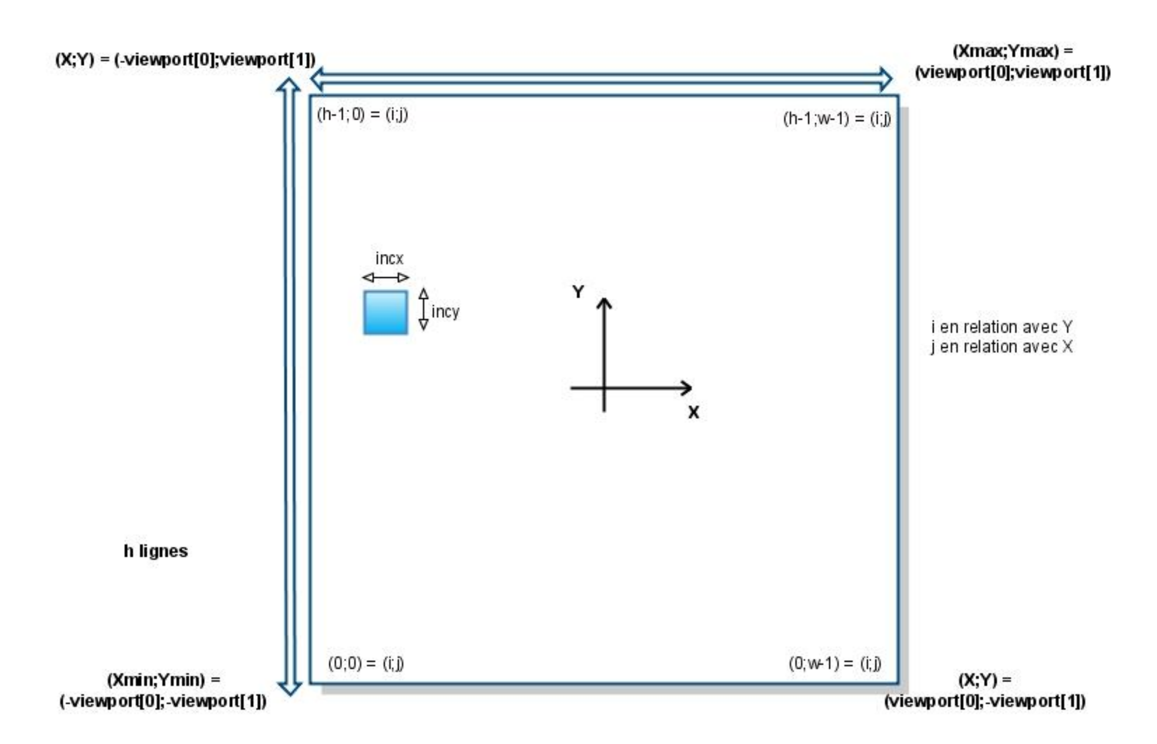


Fig. 4.1 – Découpage d'un problème de parallélisation

Dans un premier temps, nous allons paralléliser l'algorithme de la manière la plus simple. L'idée consiste à diviser l'image en "lignes" de pixels dont le traitement est attribué à plusieurs processus. Supposons que l'on dispose de NP processus. Chaque processus effectue le calcul sur une partie de l'image de dimensions $w * h$' tel que h' $= h/NP$. Afin de nous faciliter la tâche, nous considérerons que h sera toujours divisible par NP. Dès qu'un processus à terminé le traitement de l'ensemble de pixels qui lui a été alloué, il envoie son block de données au processus 0. En effet, c'est ce dernier qui gère un tableau stockant les données de l'image résultante. Il est donc à part des autres processus dans la mesure où il ne créée pas son tableau local de traitement, il affecte son calcul directement dans le tableau résultat.

On va mesurer les temps d'exécution pour chaque processus ainsi que les efficacités parallèles obtenues.

Dans un premier temps, nous mesurons le temps d'exécution du programme initial, le programme séquentiel. Nous obtenons alors le temps d'exécution suivant : $t_{seq} = 14,137s$ (il s'agissait d'une scène particulièrement complexe que nous avons lancée sur un PC relativement vieux d'où un temps d'exécution séquentiel important).

Dans un second temps, nous mesurons le temps d'exécution du programme parallèle conçu sur un réseau de huit machines. L'efficacité parallèle se calcule selon la formule suivante :

$Efficacité\,parallèle = Temps\,d'exécution\,séquentiel/(Nombre\,de\,processus{*}Temps\,d'exécution\,parallèle).$

Nombre de processus	Temps d'exécution du processus de rang								Efficacité parallèle
	0	1	2	3	4	5	6	7	
2	7.5779	7.4138							0.8923
4	7.0472	6.7534	6.9723	0.0212					0.4845
8	5.9556	0.0178	1.1744	5.5799	5.8811	1.1790	0.0267	0.0077	0.2812

Fig. 4.2 – Efficacité paralléle en découpage "naïf"

Quelques constatations :

• L'efficacité ne croît pas avec le nombre de processus mis eu jeu.

• Le processus 0 est celui dont le temps d'exécution est le plus élevé. En effet, il est chargé de récupérer les tableaux locaux des autres processus et de les stocker dans son propre tableau. Afin de déterminer l'efficacité parallèle, c'est le temps d'exécution le plus rang, donc celui du rang 0, que nous avons utilisé en tant que temps d'exécution parallèle.

• Les processus renvoyant une grande part de pixels noir possèdent un temps d'exécution moins élevé. En effet, cette couleur indique que les rayons envoyés à travers ces pixels n'ont pas recontrés d'objet et ont donc immédiatement pris la couleur du fond.

• Suite à la précédente constatation, nous concluons que l'équilibrage de charge n'est pas équilibré dans la mesure où la quantité de travail allouée n'est pas équitable entre chacun des processus.

4.2 Application sur KF-Ray

4.2.1 Equilibrage de charge dynamique

Nous avons vu précédemment que notre équilibrage de charge n'était pas réellement performant. En effet, même si chaque processus effectuait le calcul d'un même nombre de pixels, les temps de travail de chacun étaient très hétérogènes. En effet, selon la couleur du pixel, le nombre de calculs nécessaires varie (le noir nécessitant le moins de calcul). Cet algorithme n'était donc pas du tout optimisé car on n'exploitait pas les processus qui avaient terminé leur travail en premier et qui attendaient que les autres aient fini à leur tour.

L'objectif de cette nouvelle méthode est donc de réussir à répartir le temps de travail de chaque processeur de manière équitable. Trois réflexions se sont alors développées. Une idée possible était d'utiliser un équilibrage de charge statique en utilisant la notion de modulo. Chaque ligne i est répartie au processus ($i\,mod\,NP$). Cela marcherait dans le cas d'une image où les points noirs varieraient peu d'une ligne à une autre. Etant donné que l'on pouvait générer une infinité d'images différentes et qu'il est impossible de savoir à l'avance combien

de temps prendrait le calcul d'un pixel dans notre algorithme, nous avons opté pour un équilibrage de charge dynamique évoqué en cours : la méthode maître-esclave. Une autre façon plausible aurait été d'utiliser la méthode dite autorégulée.

Nous devons découper à nouveau l'image en différents blocs de tailles égales. On désigne un processus maître qui envoie aux processus esclaves les blocs à traiter. Dès que l'un d'entre eux a terminé son calcul, le processus maître lui renvoie du travail.

Néanmoins, il faut trouver le juste compromis dans la taille des blocs à traiter. En effet, si le maître distribuait le travail ligne par ligne, on perdrait trop de temps à l'envoi lors des nombreuses communications. Par ailleurs, si on envoyait des blocs trop gros, le problème précédent se reproduirait, certains processus passeraient plus de temps à calculer que d'autres.

Algorithme 4.1 Algorithme de modèle Master/Worker avec MPI

Entrées :
- Nombre de lignes : h
- Nombre de colonnes : w
- Nombre de lignes calculées par chaque processus : nlines
- Rang du processus Maitre : MASTER_NODE

Sortie :
- Tableau image des slaves : grid_slave
- Tableau final : grid

Début de l'algorithme

Initialisation de MPI
On initialise pas à 0
On définit des tags différents : TAG_REQ, TAG_DATA, TAG_END
Si le processus actuel est MASTER_NODE **Faire**
  Pour chaque processus différent de MASTER_NODE
    On envoie le premier travail avec le pas à utiliser au processus slave grâce à TAG_REQ
    On incrémente le pas
  FPour
  Tant Que pas ne sort pas du tableau final à calculer
    On reçoit le pas utilisé par n'importe quel slave grâce à TAG_REQ
    On reçoit un tableau grid_slave calculé par ce slave grâce à TAG_DATA
    On envoie un autre travail avec le pas à utiliser à ce slave grâce à TAG_REQ
    On incrémente le pas
  FTantQue
  Pour chaque processus différent de MASTER_NODE Faire (On sort du tableau final)
    On reçoit le dernier pas utilisé par n'importe quel slave grâce à TAG_REQ
    On reçoit le dernier tableau grid_slave calculé par ce slave grâce à TAG_DATA
    On envoie un message de fin de travail à ce slave grâce à TAG_END
  FPour
  On sauvegarde le tableau final grid
Sinon (les processus sont les slaves)
  Tant Que le processus n'a pas reçu de TAG_END
    On reçoit le pas à utiliser grâce à TAG_REQ
    On calcule le tableau grid_slave associé à ce pas
    On renvoit au MASTER_NODE le pas utilisé grâce à TAG_REQ
    On envoit le tableau grid_slave calculé grâce à TAG_DATA
  FTantQue
FSi

Fin de l'algorithme

On peut remarquer qu'on parle d'équilibrage de charge, pourtant le processus maître dans ce type d'algorithme ne travaille pas. Il régule uniquement les envois de travail aux esclaves et les réceptions des tableaux.

De plus, nous avons choisi par souci de rapidité à ce que le maître envoie du travail avant que les esclaves n'en demande. Ainsi, on peut économiser à chaque calcul de fait une communication, soit h' communications en moins pour h' blocs traités.

4.2.2 Résultats

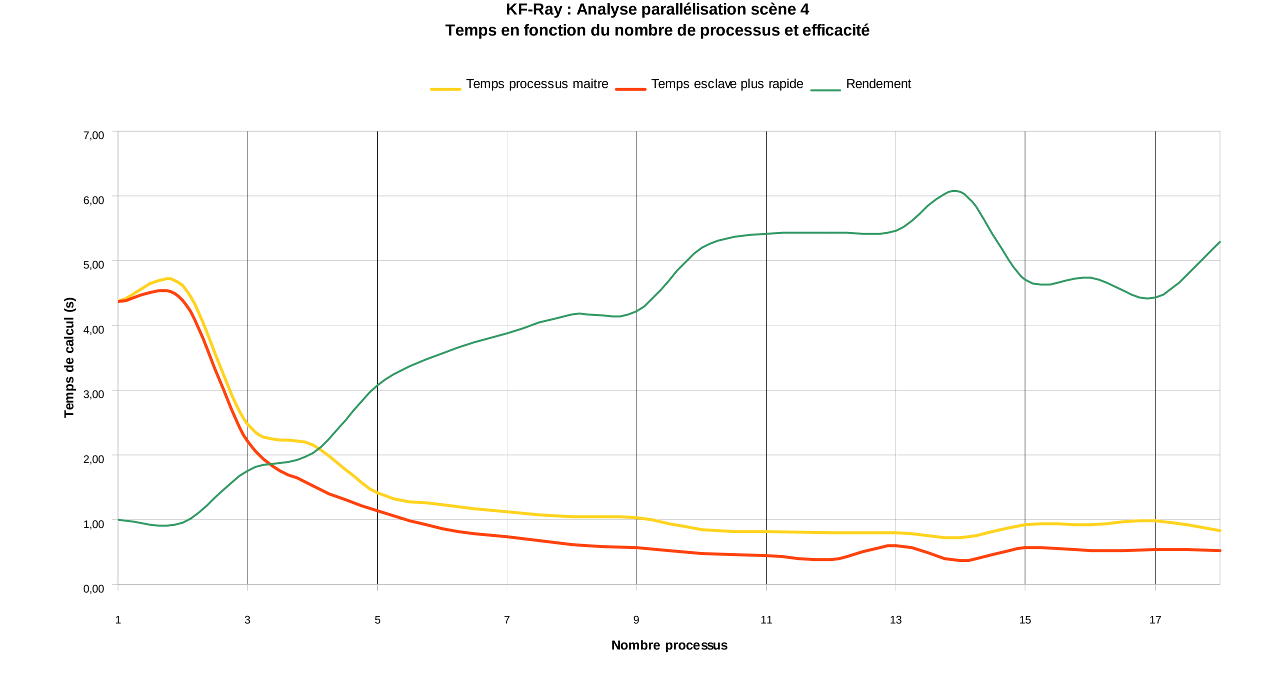


FIG. 4.3 – Graphique des performance de la parallélisation

Nous remarquons qu'étant donné que notre maître ne travaille pas, nous observons une croissance avec deux processus dû au temps de communications ajoutés au calcul sur l'esclave (qui est censé prendre le même temps vu que les machines sont homogène). Nous pouvons régler ce problème en mettant deux fois l'ip de la machine maître dans le fichier bhost.

Nous arrivons à obtenir un rendement allant jusqu'à 6 avec 14 processus signifiant que nous pouvons calculés l'image 6 fois plus vite. L'ajout d'autres ordinateurs au-delà de 14 se montre inutile pour cette scène.

4.3 Améliorations possibles de la parallélisation

Notre méthode n'est pas parfaite, loin de là. Il existe notamment plusieurs moyens de rendre cette parallélisation plus performante, moyens que nous n'avons pas implémentés faute de temps ou de réel intérêt de gain.

4.3.1 Surcoût des communications

Le premier défaut réside dans l'inactivité de l'esclave entre le moment où il envoie les données qu'il a traitées et qu'il attend en retour du processsus maître de nouvelles données. Afin de réduite ce temps de latence, on pourrait faire en sorte que le processus maître envoie une première salve de travail à tous les processus esclaves puis qu'il envoie tout de suite après une seconde salve. Ainsi, le processus esclave n'aura pas à attendre de nouvelle tâche de la part du maître, elle sera directement disponible.

Un autre aspect du problème concerne le temps que prennent les communications. Afin de réduire ce dernier, il est possible de procéder à un recouvrement des communications par les calculs. A cette fin, on pourrait utiliser le double (voir multi) buffering qui consiste à créer deux buffers : on effectue les réceptions dans les deux buffers, on effectue les calculs sur le premier, à la fin ce dernier se positionne en attente/demande de réception tandis que le second buffer effectue à son tour des calculs (après un test de fin de réception). Dans le cas de notre programme, cette amélioration n'aurait sans doute aucun effet dans la mesure où nos communications ne sont pas excessivement nombreuses ni très lourdes.

4.3.2 Faire travailler le maître

Afin d'avoir un code clair, facile nous avions décidé de ne pas faire travailler le maître. Ce dernier ne s'occupait que des communications. Une alternative consisterait donc à lui faire effectuer des calculs au même titre que ses homologues esclaves. Une première solution consisterait à modifier le code. Ceci peut se révéler fastidieux et demander un gain de temps. Une autre solution consisterait à utiliser des threads afin de donner le même travail au maître que les esclaves. Nous pourrions entre autre dans les prochaines versions utiliser OpenMP. Une dernière solution est également de mettre en place une politique de vol de travail.

Nous avons choisi d'être astucieux et de mettre deux fois l'ip de la machine maitre dans le bhost afin de la considérer à la fois maitre et esclave. Nous gagnons ainsi un processus et nous pouvons être ainsi avantagés quand nous n'avons que deux machines.

4.3.3 Auto-régulation (work-stealing)

Comme nous l'avons dit, un autre système que le modèle maître/esclave est envisageable, celui du système “auto-régulé”. Voici son principe :

- Lorsqu'un processus n'a plus de carreaux à traiter dans sa file locale de travaux en attente, il envoie de proche en proche un message de demande de travail à travers un anneau logique des processus.
- Si l'un des processus récepteurs possède encore au moins un carreau dans sa file de travaux en attente, il renvoie directement au processus demandeur la description du travail à réaliser, et le défile de sa file locale.
- Si le message de demande de travail revient à son émetteur, celui-ci sait que tous les processus sont sur le point de terminer, car ils n'ont plus de travail à lui déléguer. Il peut donc envoyer un message de demande de terminaison sur l'anneau logique, afin que tous les processus qui le reçoivent ne demandent pas de travail à leur tour mais terminent lorsqu'ils auront épuisé leurs file locale. Dès qu'un processus qui a envoyé un message de terminaison reçoit un message de terminaison (que ce soit le sien qui a fait le tour de l'anneau, ou bien celui d'un autre processus), il peut alors terminer sans le faire suivre au préalable sur l'anneau logique.

Cette solution résout le plupart de nos lacunes et serait envisageable pour une prochaine version plus performante du programme.

Conclusion

Au terme des quatres mois, KF-Ray a atteint la majorité des objectifs que nous nous étions fixés.

Notre programme génère des images d'une qualité similaire à celle de POV tout en étant plus rapide sur plusieurs machines grâce au parallélisme que nous avons introduit.

Tout utilisateur n'a pas besoin de connaissances approfondies pour sa prise en main contrairement à POV-Ray. Les options par défaut de KF-Ray lui permettent de générer une image facilement sans avoir à préciser tel modèle ou tel coefficient à utiliser.

KF-Ray propose une documentation fournie, un manuel d'utilisateur didactique, un accès facile au code source, un support utilisateur sur les sites.

Le projet reste bien entendu largement perfectible notamment au niveau des réfractions, de l'ajout de textures sous forme d'images (environment mapping) ou encore du parallélisme comme nous l'avons vu.

Finalement, ce projet, dans la continuité des projets courts, a demandé un travail d'équipe, une gestion plus délicate étant sa difficulté et sa durée accrues. Tels des développeurs de logiciels, nous avons appris à nous intéresser et à utiliser les outils standards et avancés de programmation.

Nous avons beaucoup apprécié le projet, et nous espérons continuer à produire de nouvelles releases.

Annexe A

Exemple de fichier de scène complet

```
/*
** Description de scène 1
** Scène par défaut
* /
Scene{
Width = 640;
Height = 480;
Materials = 3;
Planes = 0;
Spheres = 3;
Lights = 2;
Camera = 0.0, 0.0, -110.0;
}

//Mat jaune
Material{
Id = 1;
Type = turbulence;
RGB = 255.0, 255.0, 0.0;
Reflection = 0.5;
Specular = 1.0;
Roughness = 60.0;
Perlin = 1.0;
Bump = 0.0;
}

//Mat Cyan
Material{
Id = 2;
Type = marbel;
RGB = 0.0, 255.0, 255.0;
Reflection = 0.5;
Specular = 1.0;
Roughness = 60.0;
Perlin = 1.0;
Bump = 0.0;
}

//Mat Magenta
Material{
Id = 3;
Type = wood;
RGB = 255.0, 0.0, 255.0;
Reflection = 0.5;
Specular = 1.0;
Roughness = 60.0;
Perlin = 1.0;
Bump = 0.0;
}
Sphere{
Center = 260.0, 290.0, 0.0;
Radius = 100.0;
Material = 1;
}
Sphere{
Center = 440.0, 290.0, 0.0;
Radius = 100.0;
Material = 2;
}
Sphere{
Center = 350.0, 140.0, 0.0;
Radius = 100.0;
Material = 3;
}
Light{
Position = 640.0, 240.0, -10000.0;
Intensity = 0.5, 0.5, 0.5;
}
Light{
Position = 0.0, 240.0, -100.0;
Intensity = 0.2, 0.2, 0.2;
}
```

Annexe B

Intersection d'un rayon avec une sphère

Rayon $\overrightarrow{OP} = O_R + \lambda.\overrightarrow{u}$

- O est l'origine du repère
- O_R est l'origine de la source lumineuse
- λ est l'abscisse du rayon
- $\overrightarrow{u}$ est le vecteur directeur du rayon

Sphère $\left\|\overrightarrow{O_sM}\right\|^2 = r^2$

- O_s est le centre de la sphère
- r est le rayon de la sphère

Soit M l'intersection entre la sphère et le rayon. M vérifie : $\begin{cases} \overrightarrow{OM} = O_R + \lambda.\overrightarrow{u} \\ \left\|\overrightarrow{OM}\right\|^2 = r^2 \end{cases}$

On veut savoir où se trouve l'intersersection. λ est l'abscisse recherchée.

On résout alors :

$$(M_x + \lambda.u_x)^2 + (M_y + \lambda.u_y)^2 + (M_z + \lambda.u_z)^2 = r^2$$

Ce qui revient à résoudre :

$$(u_x^2 + u_y^2 + u_z^2).\lambda^2 + 2.(M_x.u_x + M_y.u_y + M_z.u_z).\lambda + (M_x^2 + M_y^2 + M_z^2 - r^2) = 0$$

Ceci est une équation du second ordre que l'on sait résoudre facilement.

D'où l'algorithme employé qui affecte un nouveau λ s'il a trouvé une sphère plus proche que le dernier objet intersecté :

Algorithme B.1 Fonction calcul d'intersection d'un rayon et d'une sphère

Entrées : réel λ, objet rayon envoyé, objet sphère courante.
Sorties : réel λ, booléen VRAI si intersection plus proche FAUX sinon.

Début

On affecte :
$b \leftarrow \overrightarrow{u}.\overrightarrow{O_S O_R}$
$a \leftarrow \overrightarrow{O_s O_R}$
$c \leftarrow r^{\mathbf{2}}$
On calcule le discriminant réduit$\Delta' \leftarrow b^2 - a.c$

Si $\Delta' < 0$ **Alors**
On retourne FAUX
FSi

On calcule les racines :

$\lambda_1 \leftarrow \frac{b-\sqrt{\Delta'}}{a}$ // b est négatif

$\lambda_2 \leftarrow \frac{b+\sqrt{\Delta'}}{a}$

Si $\lambda_1 > 0$ **ET** $\lambda_1 < \lambda$ **Alors**
$\lambda \leftarrow \lambda_1$
On retourne VRAI
FSi

Si $\lambda_2 > 0$ **ET** $\lambda_2 < \lambda$ **Alors**
$\lambda \leftarrow \lambda_2$
On retourne VRAI
FSi

Fin

Annexe C

Intersection d'un rayon avec un plan

Rayon $\overrightarrow{OP} = O_R + \lambda.\overrightarrow{u}$

- O est l'origine du repère
- O_R est l'origine de la source lumineuse
- λ est l'abscisse du rayon
- $\overrightarrow{u}$ est le vecteur directeur du rayon

Plan $\alpha.x + \beta.y + \gamma.z + \delta = 0$

- $\overrightarrow{n} = \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix}$ est un vecteur normale du plan
- δ est la distance du plan par rapport à l'origine

M est l'intersection du plan et du rayon ssi : $\begin{cases} \alpha.x + \beta.y + \gamma.z + \delta & = 0 \\ x & = O_{Rx} + \lambda.u_x \\ y & = O_{Ry} + \lambda.u_y \\ z & = O_{Rz} + \lambda.u_z \end{cases}$

En injectant on obtient :

$$\alpha.(O_{Rx} + \lambda.u_x) + \beta.(O_{Ry} + \lambda.u_y) + \gamma.(O_z + \lambda.uz) + \delta = 0$$

En isolant λ on obtient :

$$\lambda = \frac{-(\alpha.O_{Rx} + \beta.O_{Ry} + \gamma.O_{Rz} + \delta)}{\alpha.u_x + \beta.u_y + \gamma.u_z}$$

On en déduit un algorithme simple, qui affecte un nouveau λ s'il a trouvé un plan plus proche que le dernier objet intersecté :

Algorithme C.1 Fonction calcul d'intersection entre un rayon et un plan

Entrées : réel λ, objet rayon envoyé, objet plan courant.
Sorties : réel λ, booléen VRAI si intersection plus proche FAUX sinon.

Début

Si $\overrightarrow{u}.\overrightarrow{n} < 0$ **Alors**
On retourne FAUX
FSi

$\lambda_0 = \frac{-(\overrightarrow{n}.\overrightarrow{OO_R}+\delta)}{\overrightarrow{u}.\overrightarrow{n}}$

Si $\lambda < \lambda_0$ **OU** $\lambda_0 < 0$ **Alors**
On retourne FAUX
FSi

$\lambda = \lambda_0$

Fin

Bibliographie

[1] B. Péroche, D. Ghazanfarpour, D. Michelucci, M. Roelens. *Informatique Graphique : Méthodes et Modèles*. 1997, HERMES.

[2] J. Foley, A. van Dam, S. Feiner, J. Hughes, R. Phillips. *Introduction to computer graphics*. 1994, Addison-Wesley.

[3] N. Dodgson, A. Blackwell. *Some Mathematics for Advanced Graphics*. University of Cambridge Computer Laboratory. http ://www.cl.cam.ac.uk/teaching/1999/AGraphHCI/SMAG/

[4] Grégory Massal. *The Raytracer series* from Codermind.
http ://www.codermind.com/articles/Raytracer-in-C++-Introduction-What-is-ray-tracing.html

[5] P. Fortin. Cours et TP de programmation parallèle Polytech'Paris-UPMC.

[6] J. Levine. *Lex & Yacc*. 1994, O'Reily.

[7] C.T. Lieu. Cours de compilation Polytech'Paris-UPMC.

[8] H. Pennington. *Programmation GKT+/GNOME*. 2000, Campus Press.

[9] POV-Ray The Persistence of Vision Raytracer. http ://www.povray.org/

[10] J.B. Mouret, MM. Preche, Storde, Solt. zRCube. http ://sourceforge.net/projects/zrcube/

[11] M. Ammi. *Algorithmique Graphique*. Université Paris-Sud 11. 2009.

[13] P. Ris. *Parallélisation du lancer de rayon par évaluation dynamique de la topologie de la scène*. Thèse 1993-1996.

[14] The Free Encyclopedia. http ://www.wikipedia.org